Cryptography

Academic Year 2023/2024

Nicola Aggio 880008

Index

1 Security

Security generically refers to the **possibility** of **'protecting' information**, which is either stored in a **computer system** or transmitted on a **network**, against different types of attacks. The reason why security is so important is because there are now many computer systems that share resources (*system security*) and a wide spread of distributed systems (*network security*): notice that in this course we'll deal with both types of security. The main areas involved in this process of security are telecommunications, electrical power systems, gas and oil etc.., while some examples of threats (or attacks) that may jeopardize their security are:

- *DoS*;
- *Modified DBs*;
- *Virus*;
- *Identity theft*;
- ...

1.1 Security properties

There are many different aspects we might want to protect. We list the most important ones below. Each of them correspond to a different security property:

- **Authenticity**: an **entity** should be **correctly identified**. This may apply to different settings. For example, the *login* process allows for authenticating a user, a *digital signature* (that we will discuss) allows for authenticating the entity originating a message, and so on;
- **Confidentiality** (or secrecy): **information** should only be **accessed** (read) by **authorized entities**. *Confidentiality* involves, in turn, two aspects:
  - *Data confidentiality*, i.e. confidential **information** is **not disclosed** to unauthorized individuals;
  - *Privacy*, i.e. individuals **control** what information related to them may be collected and stored and by whom and to whom that information may be disclosed.

  In general, in order to ensure confidentiality we need to use the *"need to know"* basis for data access:
  - How do we know *who needs what* data? A possible approach would be to implement an access control that specifies *who* can access *what*, similarly to the UNIX permissions;
  - How do we know a user *is the person she claims to be*? In this case we need her identity and we need to verify this identity, and a possible approach would be of implementing an identification and authentication.

Notice that **confidentiality** is both **difficult** to **ensure**, and it is the **easiest** to **assess** in terms of success, since it is **binary** in nature (either we ensure confidentiality or not);

- **Integrity**: **information** should only be **modified** by **authorized entities**. Again, there exist two types of integrity:

  - *Data integrity*, which ensures that **information** and **programs** are **changed** only in a **specified and authorized manner**;
  - *System integrity*, which ensures that a **system** performs its **intended function**, free from unauthorized manipulation.

  Notice that there is a **difference** between integrity and confidentiality, in the sense that integrity is concerned with unauthorized *modification* (i.e. *write*) of the resources (assets), while confidentiality only deals with the *access* (i.e. *read*) to the assets. In this sense, **integrity** is **more difficult to measure** than confidentiality, also because of the fact that it is **not binary**, i.e. we can have different degrees of integrity. Finally, integrity is **context-dependent**, meaning that it refers different things in different contexts.

  Example: if we consider a quote from a politician, we can either preserve the quote (in this case we consider *data integrity*, i.e. we preserve the content of the quote), or preserve the mis-attribute (in this case we consider the *origin integrity*, i.e. we preserve the origin of the quote, the name of the politician).

- **Availability**: **information** should be **available**/**usable** by **authorized** users. This property is important to guarantee **reliability** and **safety**, since apart from attacks, availability might be lost in case of faults. This is addressed through techniques that make the system **fault-tolerant** and that we will not treat in this course. In general, we can say that an asset is available if:

  - Timely request response, i.e. whenever we ask the asset, the system provides it;
  - Fair allocation of resources, i.e. no starvation occurs;
  - Fault tolerant, i.e. no total breakdown occurs;
  - Easy to use in the intended way;
  - Provides controlled concurrency (concurrency control, deadlock control etc..).

- **Non-repudiation**: an **entity** should **not be able** to **deny** an **event**, for example, having sent/received a message. This property is crucial for e-commerce, where 'contracts' should not be denied by the parties;

There are, of course, many more properties that we do not mention here and we will not have time to address in this course. Examples are:

- Fairness of contract signing;
- Privacy;
- Anonymity;

- Unlinkability, i.e. an attacker cannot distinguish if two items are related or not;
- Accountability, i.e. tracing an event to a unique entity. Notice that there's a difference between authenticity and accountability, since the former refers to the property of being able to be verified and trusted, while the latter refers to the possibility of tracing back an action to an entity.

1.2 Typical attacks

We assume information is flowing from a source to a destination. For example, reading data is a flow for the data container (e.g. a file) to a user while writing (or modifying) is a flow from a user to the file system, an so on.

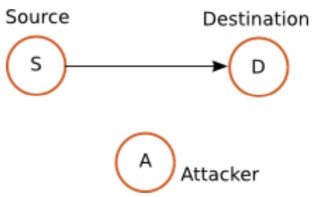


Figure 1: Expected information flow

Malicious users might try to subvert the above properties in many different ways. We now give very general classification depending on how an attacker might interfere on the expected flow of information.

1.2.1 Interruption

In this case the **attacker stops the flow of information**. This is typically a **DoS** that makes the system/network unusable:

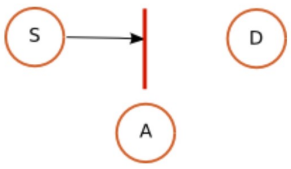


Figure 2: The attacker interrupts the flow of information

An interruption breaks system **integrity** and **availability**, both because the message is not arriving to the destination. In general, an interruption is quite easy to notice, and some examples are DoS, canceling programs or data files, destruction of part of the hardware etc..

1.2.2 Eavesdropping (or interception)

In this case the **attacker gets unauthorized access to information**. This can be depicted as an additional flow towards the attacker:

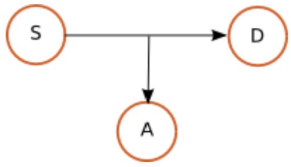


Figure 3: The attacker intercepts information

Notice that in this case the attacker is **not modifying the information**, so this attack breaks the **confidentiality** of the system, since the information is accessed by unauthorized users. Differently from interruptions, interceptions are **quite difficult to detect**, and some examples are represented by unauthorized copies of programs or files or the interception of data flowing in the network (e.g. a credit card number).

1.2.3 Modification

In this case the **attacker modifies information**. To modify information, the attacker first **intercepts** it:

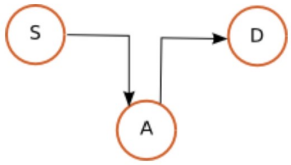


Figure 4: The attacker modifies information

This attack breaks **integrity** and **confidentiality**, because the information is modified, thus accessed, by unauthorized users. Some examples involve unauthorized change of values (e.g. of a DB), of a program, or of data flowing in a network.

1.2.4 Forging (or falsification)

In this case the **attacker introduces new information**. This is usually related to **impersonation**, since the attacker lets the destination believe the information is coming from the honest source:

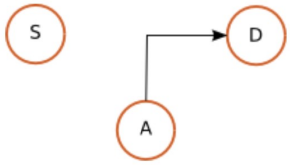


Figure 5: The attacker forges new information

This attack breaks **authenticity** (since the attacker is not correctly identified), **accountability** (since it is not possible to trace the event to the attacker) and **integrity** (since we generate a message from nothing). Examples of forging involves the addition of new messages in the network, or the addition of a record in the DB.

In general, we distinguish **two types of attacks**, as shown in Picture 1.2.4.

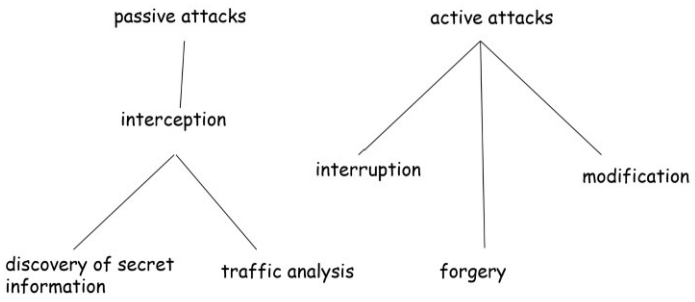


Figure 6: Passive and active attacks

1.2.5 Examples

We give two simple examples of attacks to show how the general scenarios above apply.

1. Suppose a bank B is using the following simple protocol to allow a bank transfer from user Alice (A):

   $$A \to B : \text{sign\_}A(\text{“please pay Andy 1000 Euros”})$$

   ,where sign_A is some “signature” mechanism to ensure that the message really comes from Alice. We thus assume that the attacker cannot generate valid signed messages from Alice. However, it is always possible for the attacker to intercept the whole message and repeat it as many times as he wants, without modifying it. This attack, called **replay**, consists of an **interception** plus **forging** (in this case a very simple one as the message is just re-sent as is):

   Figure 7: The attacker Andy intercept the signed message and then replays it

   In this way Andy gets the bank transfer as many times as he wants by just resending message M;

2. A *Trojan Horse* is a program that seems to behave as expected but incorporates malicious code (such as the Greek force hidden inside the mythological Trojan Horse as described in the Virgil's Aeneid). These programs are usually obtained by modifying existing, honest programs. This is in fact an example of modification attack in which S is the web site where the honest program H can be downloaded and D is the final user. The attacker downloads the honest program, modifies it and sets up a fake download site where D will download the *Trojan Horse* program.

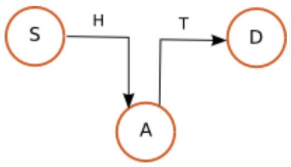


Figure 8: The honest program H is modified into a Trojan T

Such a program apparently works normally, however it changes, e.g., all the access rules of the users that is executing it. In this case is an attack to confidentiality and integrity.

2 Classical cryptography

The word *cryptography* comes from the Greek, and it means "hidden writing", and it can be defined as a powerful **tool** to **protect information**, especially when this is exposed to **insecure environments** such as the Internet, or when the system does not support sufficient protection mechanisms. Historically, it mainly aimed at providing **confidentiality**, i.e., protecting from unauthorized access.
Intuitively, cryptography amounts to **transforming** a **plaintext** into a **ciphertext** so that unauthorized users cannot easily reconstruct the plaintext. In other words, cryptography essentially consists of two phases:

1. *Encryption*, i.e. the transformation of a plaintext (or message) into a ciphertext, by using some rules;
2. *Decryption*, i.e the reconstruction of the plaintext starting from the ciphertext.

In general, the **encryption** must satisfy two **properties**:

1. It has to be **simple** and **fast**;
2. The **decryption** should be **unfeasible** for an attacker, i.e. not solvable in a finite time.

2.1 Encryption using a shared key

For what regards the encryption, we have **two** possible **solutions**:

1. **Only** the **sender** (Alice) and the **receiver** (Bob) know the **encryption algorithm**, following the schema of Picture 1.

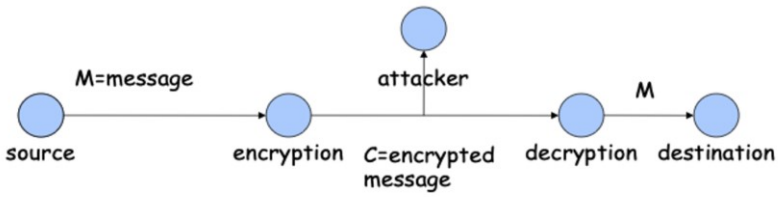


Figure 9: First encryption strategy

   An example of this type of encryption is provided by the *Enigma machine*, adopted by the German militaries during World War II;

2. The **encryption algorithm is known**, even by the attacker, but Alice and Bob share some information that is not accessible by the attacker (i.e. it travels trough a secure channel), the **encryption key**.

   As we can see, in this case both the **encryption** and the **decryption** involve the **secret key** adopted from the source and the destination, and it is usually better than the first strategy, since:

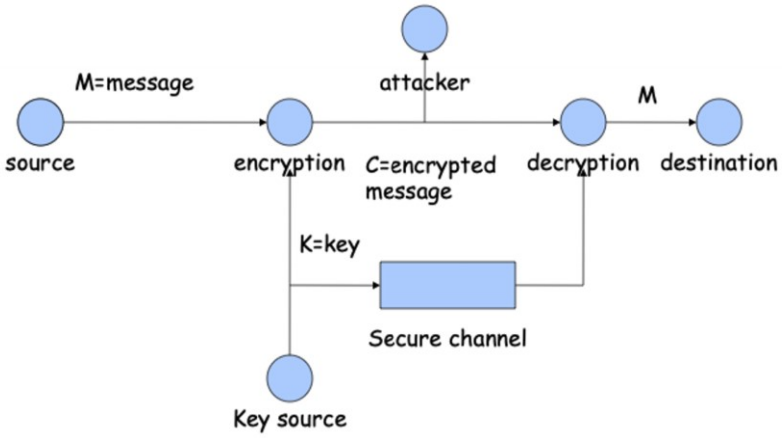


Figure 10: Second encryption strategy

- It is **simpler** to **distribute** only one **key**, rather than an entire encryption algorithm;
- If there is an attack, it is **easier** to **change key** instead of a whole algorithm.

For these reasons, this **second strategy** is **preferred**.

In general, a **cryptosystem** (or a **cipher**) can be defined as a quintuple (P, C, K, E, D), where:

- P is the set of **plaintexts**, i.e. the set of messages we want to share;
- C is the set of **ciphertexts**, i.e. the result of the encryption applied to the plaintexts;
- K is the set of **keys**;
- $E : k \times P \to C$ is the **encryption function**, which takes as input a key and a plaintext, and produces in output an cyphertext;
- $D : k \times C \to P$ is the **decryption function**, which takes as input a key and a ciphertext, and produces in output a plaintext. Notice that in this case, the input C of the decryption function represents the output of the encryption function.

If we consider $x \in P$ (plaintext), $y \in C$ (ciphertext) and $k \in K$, we write $E_k(x)$ and $D_k(y)$ to denote $E(k, x)$ and $D(k, y)$, i.e. the encryption and decryption under the key k of x and y, respectively. In this sense, we require that:

1. $D(E_k(x)) = x$, i.e. **decrypting a ciphertext** with the right key gives the **original plaintext**;
2. Computing k or x given a ciphertext y (which is what the attacker sees, as shown in Picture 2) is **unfeasible**, meaning that it is so complex that it cannot be done in a reasonable amount of time.

It is important to underline the fact that all the invented ciphers satisfies (1), but some of them do not satisfy (2), as we will see in the following sections.

2.2 Monoalphabetic ciphers

We now study some of the most important monoalphabetic ciphers, i.e. ciphers in which the **letters** of the plaintext are **mapped** to ciphertext letters based on a **single alphabetic key**.

2.2.1 Caesar cipher

This is probably the **simplest** and most famous cipher, due to Julius Caesar. The idea is very simple: each **letter** of a message is **substituted** with the one that is **3 positions next** in the alphabet. So, for example, 'A' is replaced with 'D' and 'M' with 'P'. The substitution can be represented as follows:

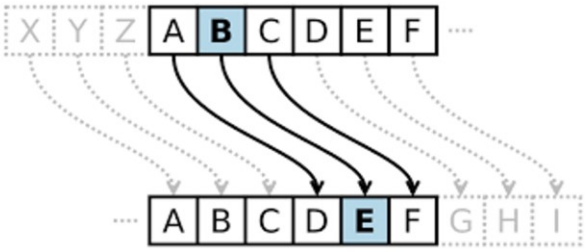


Figure 11: Caesar cipher

, meaning that each letter in the top alphabet is substituted with the corresponding one in the bottom (rotated) alphabet. For example, the word HOME would be encrypted as KRPH. To **decrypt** it is enough to apply the **inverse substitution**.

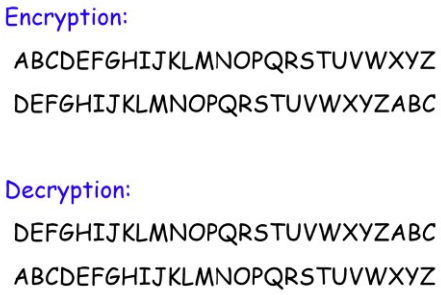


Figure 12: Caesar cipher: encryption and decryption

In this case:

1. The encryption is given by the substitution with the letter in 3 positions next in the alphabet;
2. The key is 3.

Example: we want to decrypt BHV BRX PDGH LW, and we obtain YES YOU MADE IT.
Despite being extremely simple, the Ceasar cipher is clearly **insecure** for many different reasons. First of all, **once the cipher has been broken** any **previous** exchanged

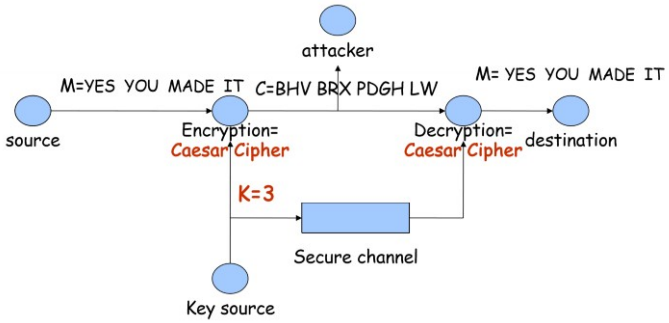


Figure 13: Caesar cipher: example

message is also **broken**. This is due to the fact that this cipher always works in the same way. There is a famous **principle** in cryptography, due to Auguste Kerckhoffs, that tells that a **cipher** should remain **secure even** if the **algorithm** becomes **public**. This is achieved by **parametrizing** ciphers on a **key**. The key can be changed and is assumed to be the only secret. This is of course fundamental if we want a cipher to scale and be used by millions of users.
Other Kerckhoffs rules (1883) are:

1. The system should be, if not theoretically unbreakable, **unbreakable in practice**;
2. The **design** of a system should **not** require **secrecy**, and compromise of the system should not inconvenience the correspondents;
3. The **key** should be **memorable** without notes and should be easily changeable;
4. The cryptograms should be transmittable by a telegraph;
5. The apparatus or documents should be portable and operable by a single person;
6. The system should be easy, neither requiring knowledge of a long list of rules nor involving mental strain.

2.2.2 Shift ciphers

We now consider a variant of the cipher, called **shift cipher**, which is parametrized on a key k, that we assume to range from 0 to 25. Intuitively, k represents the number of positions in the alphabet that we **shift** each letter of (since we have 26 letters, we can perform 26 shifts, including the shift of 0 positions). Notice that in this case:

$$P = C = K = \mathbb{Z}_{26}$$

, i.e. the plaintexts, the ciphertexts and the keys are the set of integers from 0 to 25. Moreover:

- $E_k(x) = (x + k) \mod 26$;
- $D_k(y) = (y - k) \mod 26$, where $y = E_k(x)$;

Notice that **Caesar cipher** represents a **subcase** of **shift cipher** when $k = 3$.

ABCDEFGHIJKLMNOPQRSTUVWXYZ
KLMNOPQRSTUVWXYZABCDEFGHIJ

Figure 14: Shift cipher: example

For example $k = 10$ gives the following substitution (notice that the bottom alphabet is now shifted to the left by 10 positions):
Does the first property hold? We have to check whether

$$D_k(E_k(x)) = x$$

We have that:

$$D_k((x+k) \mod 26) = [(x+k) \mod 26-k] \mod 26 = x+(k-k) \mod 26 = x \mod 26 = x$$

, so the **first property** holds. The previous result depends from the fact that $\mathbb{Z}_{26}$ is a group under the addition (not under multiplication). We recall that a group $< G, * >$ is a set G together with a (closed) binary operation $*$ on G s.t.:

- The operator is associative, i.e. $(x * y) * z = x * (y * z)$;
- There is an element $e \in G$ s.t. $a * e = e * a = e$, the identity element;
- For every $a \in G$, there is an element $b \in G$ s.t. $a * b = e$, the inverse element.

Notice that $\mathbb{Z}_{26}$ is not closed under multiplication since there's no multiplicative inverse for every element in $\mathbb{Z}_{26}$.
A possible **attack** for shift cipher is the **brute force attack**, which consists in trying all the possible 26 keys (i.e. each possible shift): thus, the **second property does not hold**, since the cipher is **not secure** if the algorithm becomes public.

2.2.3 Substitution cipher

A possible method for overcoming the previous limitation is to consider a **generic permutation of the alphabet**, so consider a k as a random permutation. Formally,

$$P = C = \mathbb{Z}_{26}$$

and $k = \{\rho | \rho$ is a permutation of 0,..,25$\}$. Moreover:

- $E_k(x) = \rho(x)$;
- $D_k(y) = \rho^{-1}(y)$.

An example is provided in Picture 2.2.3.
Let's see if this cipher satisfy the properties:

ABCDEFGHIJKLMNOPQRSTUVWXYZ
SWNAMLXCVJBUYKPDOQERIFHGZT

Figure 15: Substitution cipher: example

1. $D_k(E_k(x)) = x$. We have that

$$D_k(\rho(x)) = \rho^{-1}(\rho(x)) = x$$

, so the **first property is satisfied**;

2. Computing k or x given a ciphertext y is unfeasible. Is the brute force technique still possible in this case? Well, we have 26! possible keys (all the possible permutations of 26 elements), which is approximately $4 * 10^{26} > 2^{88}$, which would be **unfeasible** even with powerful parallel computers.

However, this cipher is characterized by an important **limit**, which is of being a **monoalphabetic cipher**, meaning that it **maps** a **letter always** to the very **same letter**. This preserves the **statistics** of the plaintext and makes it possible to **reconstruct the key** by observing the statistics in the ciphertext. For example, vowels e,a,o,i will be easy to identify as they are much more frequent than the other letters.
In this sense, if the attacker knows the used **cipher** (e.g. monoalphabetic substitution cipher) and the **language** of the plaintext (e.g. Italian), even without knowing the plaintext and the key he could be able to decrypt the plaintext, by exploiting the statistics of the language of the plaintext. For example, if he knows that the letter S appears 0 times, while letter C appears 15 times, then he can infere that letter C is a transformation of letter A, since it appears a lot of times. For each language we can build a chart of the most frequent letters/bigrams/trigrams, as showed in Picture 2.2.3.
Other deductions can be made from the fact that, for example, in Italian the frequency of letter I and L increases at the beginning of the sentences, while the one of A,E,I,O,U at the end of the words etc..
In this sense, a possible approach for decrypting a monoalphabetic substitution cipher could be the following:

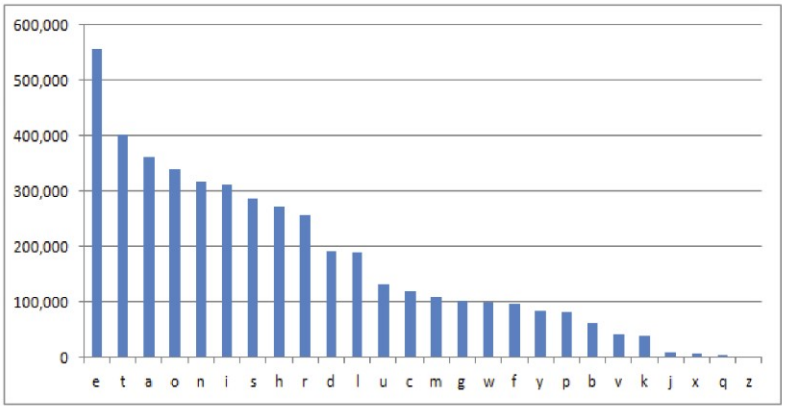
1. We **order** the letters of the ciphertext into decreasing frequencies;
2. We **substitute** with letters in decreasing order as in the corresponding tables (depending on the language), by eventually exchanging letters with similar frequencies and by also exploiting digraphs and trigraphs frequently used,

2.3 Polyalphabetic ciphers

We have seen that **monoalphabetic** ciphers are prone to **statistical attacks**, since they preserve the statistical structure of the plaintext. To overcome this issue, it is important that the same plain symbol is not always mapped to the same encrypted symbol. When this happens the cipher is called **polyalphabetic**.

TABELLA 2.1 *Frequenze assolute e percentuali delle lettere singole (le frequenze percentuali sono arrotondate alla terza cifra decimale).*

A	1714	0,114
B	160	0,011
C	537	0,042
D	566	0,038
E	1658	0,111
F	141	0,009
G	272	0,018
H	160	0,011
I	1563	0,104
L	966	0,064
M	436	0,029
N	966	0,064
O	1486	0,099
P	421	0,028
Q	85	0,006
R	978	0,065
S	771	0,051
T	1024	0,068
U	528	0,035
V	343	0,023
Z	123	0,008
Totali	14.998	0,998



Single Letter Frequencies for Modern English Spelling based on a 1 million-word sample

Figure 16: Language statistics

2.3.1 Vigenére cipher

This simple polyalphabetic cipher works on **"blocks"** of m letters with a key of length m. For example, if we consider the text "THISISAVERYSECRETMESSAEG" and the key "FLUTE", the plaintext is splitted into blocks of length 5, and the key FLUTE is repeated as necessary and used to encrypt each block, as showed in Picture 2.3.1.

```
THISISAVERYSECRETMESSAGE +
FLUTEFLUTEFLUTEFLUTEFLUT =
YSCLMXLPXVDDYVVJEGXWXLAX
```

Figure 17: Vigenére cipher: example

Formally, $P = C = K = \mathbb{Z}_{26}^m$, where $\mathbb{Z}_{26}^m$ is $\mathbb{Z}_{26} \times \mathbb{Z}_{26} \times \ldots \times \mathbb{Z}_{26}$, m times. Encryption and decryption are defined as follows:

- $E_{k1,..,km}(x_1, .., x_m) = (x_1 + k_1, .., x_m + k_m) \mod 26$;

- $D_{k1,..,km}(y_1, .., y_m) = (y_1 - k_1, .., y_m - k_m) \mod 26$

In the example above, $k_1 = \text{F}$, $k_2 = \text{L}$ etc..
The first **strength** of this cipher is that since the **number of possible keys** is 26^m (i.e. the number of possible keys is given by the total number of possible sequences of m letters), for m big enough this **prevents brute force attacks**. Another strength is given by the fact that **one letter** is **not always mapped** to the **same one** (unless the are at a distance that is multiple of m). For example the fist two letters "I" are encrypted as "C" and "M", respectively. While the "S" in position 6 and the last one are both encrypted as "X" using the "F" of "FLUTE". They are, in fact, at distance 15 which is a multiple of 5. Thus, the **histogram** of the **frequencies** is **flat**, and the flatness increases with the increase of the key length.

Breaking Vigenére cipher: the Friedman method Even if Vigenére cipher hides the statistic structure of the plaintext better than monoalphabetic ciphers, it still preserve most of it. There are two famous methods to break this cipher. The first is due to Friedrick Kasiski (1863) and the second to Wolfe Friedman (1920). We illustrate the latter since it is more suitable to be mechanized. Both are based on the following steps:

1. Recover the **length** m of the **key**. The intuition is that once we know m, we know that at distance m we'll find the same letter of the key, thus the letters at distance m form a monoalphabetic cipher;
2. Recover the **key**.

STEP 1: recover m
The Friedman method uses statistical measures to recover the length m of the key. In particular, we consider the **index of coincidence**, which is defined as:

$$I_c(x) = \frac{\sum_{i=1}^{26} f_i(f_i - 1)}{n(n-1)} \approx \sum_{i=1}^{26} p_i^2$$

, where:

- x represents the text for which the index of coincidence is computed;
- f_i represents the frequency of the i-th letter in the text;
- n is the length of the text;
- p_i represents the probability of the i-th letter, and it is computed as $p_i = \frac{f_i}{n}$.

Intuitively, the index of coincidence measures the **probability** that **two letters**, chosen at **random** from the text x, are the **same**. Indeed, the index is computed by multiplying the probability that the first letter is the i-th ($\frac{f_i}{n}$) and the probability that the second letter is the i-th ($\frac{f_i - 1}{n-1}$): in this case we subtract 1 since the first letter has been already fixed).
Example: we compute the index of coincidence of the text "the index of coincidence". We have that:

$$\frac{c(3*2)+d(2*1)+e(4*3)+f(1*0)+h(1*1)+...}{21*20}=0.0809$$

If we consider the random text "bmqvszfpjtcsswgwvjlio", the index has value 0.0286: as we can see, the index of coincidence of a random sentence is smaller that the one of a normal sentence.
More specifically, notice that the **value** of the index is **minimum**, with value $1/26 \approx 0.038$, for texts composed of **letters** chosen with **uniform probability** $1/26$ while it is **maximum**, with value 1, for texts composed of just a **single letter** repeated n times. It is, in fact, a **measure** of how **non-uniformely letters** are **distributed** in a **text**. For this reason, each **natural language** has a characteristic **index of coincidence**: for English the value is approximately 0.065.
In general, using frequencies analysis, we saw that if **frequencies** are **flat** we have a **polyalphabetic cipher**, if we have **peaks** and valleys of frequencies we have a **monoalphabetic cipher**. If the **value** of the **index of coincidence** is **minimum** (≈ 0.038) we have a **polyalphabetic cipher**, if it is ≈ 0.065 we have a **monoalphabetic cipher** (same as English, just a permutation of letters).
Now the question is: how do we recover the length m of the key using the Friedman method, and in particular the concept of index of coincidence? Well, the idea is to find the length by brute-forcing, following this algorithm:

```
m = 1
LIMIT=0.06 # this is to check that ICs are above 0.06 and thus close to 0.065
found = False
while(not found):
    sub = subciphers() # takes the m subciphertexts sub[m] obtained by selecting one letter every m
    found = True
    for i in range(0,m): # compute the Ic of all subtexts
        if Ic(sub[i]) < LIMIT:
            # if one of the Ic is not as expected try to increase length
            found = False
            m += 1
            break
# survived the check, all Ic's are above LIMIT
output(m)
```

Figure 18: Friedman method: estimating m

As we can see, the idea of the algorithm is the following:

1. We **initialize** the value of m to 1 (a value $m = 0$ does not make sense);
2. We consider the m **sub-ciphertexts** obtained by selecting one letter every m (e.g. for $m = 2$ we get the two sub-ciphertexts composed of only the letters in odd positions and even positions, respectively);
3. Then, we compute the **index of coincidence** of all the sub-ciphertexts, increasing the value of m if the current index has a value smaller than 0.065 (the IC of the English language). We require that the index of coincidence of all the subtexts is close to the characteristic index of the plaintext language. Typically, the bigger is the index the higher is the probability that the frequencies of the letters are close to

the one of the plaintext language. It is thus enough to choose a lower bound such as 0.06 and check that the ICs are above that;

4. The loops stops when $IC(\text{sub-ciphertext}) \geq 0.065$.

STEP 2: recover the key Once we have found the length m of the key, we need to **find the key**. We consider a **text** composed of **letters** at **distance** m from the first one, and the ones at distance m from the second one. They have different shifts, how can we find the **relative right shift**? An example is provided in Picture 2.3.1.

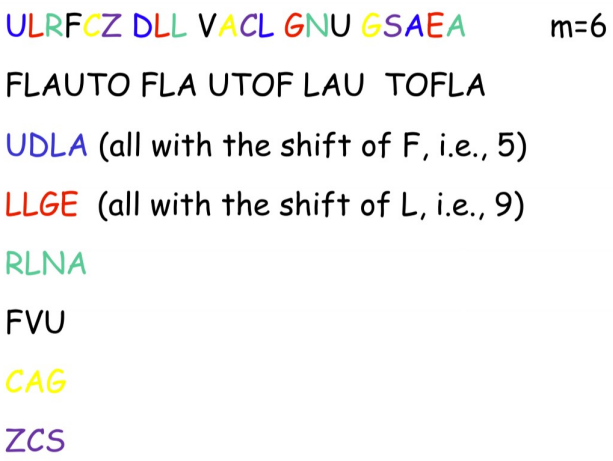


Figure 19: Examples of shifts

Our goals now are:

1. Determine the **shift** between the first letter of the key and the other $m-1$ letters;
2. Determine the **first letter** (brute force on 26 possible letters) and, by exploiting the shifts we've just computed, determine the remaining letters of the key.

We find the **relative shift** between two sub-ciphers by using the **mutual index of coincidence**, which is defined as:

$$MI_c(x, x') = \frac{\sum_{i=1}^{26} f_i f'_i}{nn'} = \sum_{i=1}^{26} p_i p'_i$$

, where:

- x represents the first text for which the mutual index of coincidence is computed;
- x' represents the second text for which the mutual index of coincidence is computed;
- f_i represents the frequency of the i-th letter in the text x;
- f'_i represents the frequency of the i-th letter in the text x';
- n is the length of the text x;
- n' is the length of the text x';

- p_i represents the probability of the i-th letter in text x, and it is computed as $p_i = \frac{f_i}{n}$;
- p'_i represents the probability of the i-th letter in text x', and it is computed as $p'_i = \frac{f'_i}{n'}$.

Intuitively, the mutual index of coincidence represents the **probability** that **two letters** taken from two texts x and x' are the **same**.
The idea is to **shift one sub-cipher** until the **mutual index** of coincidence with the first sub-cipher becomes **close** to the one of the plaintext language. When this happens, we know that the applied shift is the relative shift between the two sub-ciphers and, consequently, between the corresponding letters of the key. This is encoded in the following algorithm. In fact, what we do, is to select the relative shift that maximizes the mutual index of coincidence.

```
key = [] # empty list
for i in range(0,m): # for any letter of the key
    k = 0      # current relative shift
    mick = 0   # maximum index so far (we start with 0)
    for j in range(0,26): # for any possible relative shift
        # compute the mutual index of coincidence between the first subcipher
        # sub[0] and the i-th subcipher shifted by j
        mic = MIc(sub[0], shift(j,sub[i]))
        if mic > mick: # if it is the biggest so far
            k = j      # we remember the relative shift
            mick = mic # ... and the new maximum
    key.append(k)      # we append to the list the shift we have found
```

Figure 20: Friedman method: finding the key

We **repeat** this **for each letter of the key**, and we obtain the **list of relative shifts**. For example key = [0,4,6,3,9] means that the second letter of the key is equal to the first plus 4 while the third is the first plus 6 and so on. The final step is to try all the possible 26 first letter of the key, giving 26 possible keys (this step could be avoided computing the MIc with a reference text written in the plaintext language).

2.3.2 Properties

In general, polyalphabetic ciphers are more **difficult** to be **decrypted**, but still are **not strong enough** (we presented a method for attacking Vigenére cipher).

2.4 Known-plaintext attacks

So far, we have considered **attackers** that **only know the ciphertext** y and try to **find** either the **plaintext** x or the **key** k. In practice, it is often the case that an **attacker** can **guess part of the plaintext**. Think of encrypted messages: a message always have a standard header in a certain format and it is often easy to guess part of the information in it. Thus if a message is split into blocks which are encrypted under the same key, it is reasonable to assume that an attacker can deduce part of the plaintext. Also, if a key is

reused to encrypt many plaintexts, it can occur that in the future one of the plaintext is leaked (because its security is no more relevant). This gives the attacker knowledge of a pair (x, y) **plaintext, ciphertext**.
For these reasons, in cryptography we often consider so-called **known-plaintext attacks**, i.e., we assume the attacker knows some pairs $(x', y'), (x'', y''), ..$ of plaintexts/ciphertexts. The challenge, given y, is to find the relative x or the k. We illustrate this kind of attacks on a classical cipher.
In general, the possible attacks we can consider are:

- **Ciphertext-only attacks**: in a ciphertext-only attack the **attacker** is **assumed** to have access only to a set of **ciphertexts** (e.g., monoalphabetic ciphers, polyalphabetic ciphers). In this case the **limit** is that it is **easy to find** the **correspondence** between **letters** in the **plaintext** and in the **ciphertext**;
- **Known-plaintext attacks**: the attacker knows some **pairs** $(x', y'), (x'', y''), ..$ of plaintexts/ciphertexts;
- **Chosen-plaintext attack (CPA)**, which presumes that the attacker can ask and obtain the **ciphertexts** for **given plaintexts**;
- **Chosen-ciphertext attack (CCA)**, where the cryptanalyst can gather information by obtaining the **decryptions** of **chosen ciphertexts**.

2.4.1 The Hill cipher

This cipher is polyalphabetic and generalizes the idea of Vigenére by introducing **linear transformations** of blocks of plaintext.
Formally, $\mathcal{P} = \mathcal{C} = \mathbb{Z}_{26}^m$, while $\mathcal{K} = \{K \mid K \text{ is an invertible mod 26 } m \times m \text{ matrix}\}$.
Encryption and decryption are defined as follows:

- $E_k(x_1, .., x_m) = (x_1, .., x_m)K \mod 26$, i.e. we multiply the message and the matrix, modulo 26;
- $D_k(y_1, .., y_m) = (y_1, .., y_m)K^{-1} \mod 26$, i.e. we multiply the encrypted message and the inverse of the matrix, modulo 26.

Example: let us assume $M = \text{message} = (x_1, x_2) = (5, 9)$, and $K = \begin{bmatrix} 5 & 11 \\ 8 & 3 \end{bmatrix}$
To **encrypt** M, we have to:

- Take $M = (x_1, .., x_m)$ and K which is a **matrix**;
- Compute $(x_1, ..., x_m)K \mod 26$.

Thus, $E_k(5, 9) = (5, 9) * \begin{bmatrix} 5 & 11 \\ 8 & 3 \end{bmatrix} \mod 26 = (25 + 72, 55 + 27) \mod 26 = (19, 4)$
To **decrypt** M, we have to:

- Compute the **inverse** of K, i.e., K^{-1} (K is invertible);
- Compute $(y_1, .., y_m)K^{-1} \mod 26$.

The **inverse** of K is computed as:

$$K^{-1} = \det^{-1}(K) \begin{bmatrix} 3 & -11 \\ -8 & 5 \end{bmatrix} \mod 26 = \det^{-1}(K) \begin{bmatrix} 3 & 15 \\ 18 & 5 \end{bmatrix} \mod 26$$

Now,

$$\det(K) = (15 - 88) \mod 26 = 5$$

, and to compute $\det^{-1}(K)$ (i.e. the inverse mod 26 of 5) we need to find a number in the interval [0,25] that multiplied by 5 mod 26 gives 1. In this case the number is 21 ($5 * 21 \mod 26 = 1$). Thus, $\det^{-1}(K) = 21$. Notice that it is not always the case that the multiplicative inverse modulo exists. We will discuss this more in detail when introducing public key cryptography and RSA. We now have to solve:

$$K^{-1} = \det^{-1}(K) \begin{bmatrix} 3 & 15 \\ 18 & 5 \end{bmatrix} \mod 26 = 21 \begin{bmatrix} 3 & 15 \\ 18 & 5 \end{bmatrix} \mod 26 = \begin{bmatrix} 63 & 315 \\ 378 & 105 \end{bmatrix} \mod 26 = \begin{bmatrix} 11 & 3 \\ 14 & 1 \end{bmatrix}$$

Thus,

$$D_k(19, 4) = (19, 4) * \begin{bmatrix} 11 & 3 \\ 14 & 1 \end{bmatrix} \mod 26 = (265, 62) \mod 26 = (5, 9)$$

Assume, now, that the **attacker** knows (at least) m **pairs** of plaintexts/ciphertexts, where m is the block length, and his goal is to **find** the relative x or k given y. We know that:

$$(y_1^1, .., y_m^1) = (x_1^1, .., x_m^1)K \mod 26$$

$$(y_1^m, .., y_m^m) = (x_1^m, .., x_m^m)K \mod 26$$

, where each x^i represents a plaintext, and each y^i represents a ciphertext. Moreover, the previous system of equations can be written as:

$$Y = XK \mod 26$$

, where $X = \begin{bmatrix} x_1^1 & \dots & x_m^1 \\ \dots & \ddots & \dots \\ x_1^m & \dots & x_m^m \end{bmatrix}$ and $Y = \begin{bmatrix} y_1^1 & \dots & y_m^1 \\ \dots & \ddots & \dots \\ y_1^m & \dots & y_m^m \end{bmatrix}$.
It is now clear that if X^{-1} exists, we obtain:

$$X^{-1}Y \mod 26 = X^{-1}XK \mod 26$$

, but $X^{-1}X = 1$, so $K = X^{-1}Y \mod 26$.
The **Hill cipher** is a **linear transformation of a plaintext block** into a **cipher block**. The above attack shows that this kind of **transformation** is **easy to break** if enough **pairs** of **plaintexts** and **ciphertexts** are **known**. **Modern ciphers**, in fact, always contain a **non-linear component** to prevent this kind of attacks.

Example: assume that the attacker has the pairs $(5, 9) \to (19, 4)$ and $(2, 5) \to (24, 11)$, and $m = 2$. How can we find K? Firstly, we need to compute X^{-1}: if this matrix exists, we can derive K from the formula above.

$$\begin{aligned} X^{-1} &= \det(X)^{-1} \begin{bmatrix} 5 & -9 \\ -2 & 5 \end{bmatrix} \mod 26 \\ &= \det(X)^{-1} \begin{bmatrix} 5 & 17 \\ 24 & 5 \end{bmatrix} \mod 26 \\ &= 15 \begin{bmatrix} 5 & 17 \\ 24 & 5 \end{bmatrix} \mod 26 \\ &= \begin{bmatrix} 23 & 21 \\ 22 & 23 \end{bmatrix} \end{aligned}$$

Then, $K = X^{-1}Y \mod 26 = \begin{bmatrix} 23 & 21 \\ 22 & 23 \end{bmatrix} \begin{bmatrix} 19 & 4 \\ 24 & 11 \end{bmatrix} \mod 26 = \begin{bmatrix} 5 & 11 \\ 8 & 3 \end{bmatrix}$

2.5 Euclidean algorithm

We now consider an **algorithm** for computing in an **efficient** way the operation of *inverse modulo n*. We begin by introducing the **Euclidean algorithm**, which is used to computed $\gcd(c, d)$ represented in Picture 2.5.

```
def Euclid(c,d):
    while d != 0:
        tmp = c % d
        c = d
        d = tmp
    return c
```

Figure 21: Euclidean algorithm

In this case, the idea is that, to compute $\gcd(c, d)$, when $d \neq 0$, we can compute $\gcd(d, c \mod d)$, since it can be easy seen that they're the same.
Example: $\gcd(5, 15) = 5$
This algorithm terminates in $O(k^3)$, given that the **number of iterations** is $O(k)$. This latter fact can be proved by observing that every 2 steps we at least halve the value d. Assume by contradiction that after one step this is not true, i.e., $c \mod d > \frac{d}{2}$. Next step will compute $d \mod (c \mod d) = d - (c \mod d) < \frac{d}{2}$ giving the thesis. We know that halving leads to a logarithmic complexity, i.e., linear with respect to the number of bits k.
We now **extend** the **algorithm** so to compute the **inverse modulo** d whenever $gcd(c, d) = 1$. We substitute the computation of $c \mod d$ with:

$$q = c/d$$

, i.e. an integer division, and

$$\text{tmp} = c - qd$$

, which represents the operation $c \mod d$.
Example: $12 \mod 5 = 2$, $q = \frac{12}{5}$ and $\text{tmp} = 12 - 2 * 5 = 2$.
We add two extra variables e and f and we save the initial value of d in d_0, as follows:

```
def EuclidExt(c,d):
    d0 = d
    e = 1
    f = 0
    while d != 0:
        q = c//d        # integer division
        tmp = c - q*d   # this is c % d
        c = d
        d = tmp
        tmp = e - q*f   # new computation for the inverse
        e = f
        f = tmp
    if c == 1:
        return e % d0   # if gcd is 1 we have that e is the inverse
```

Figure 22: Extended euclidean algorithm

Example: the inverse between 5 and 17 can be computed using $\text{EuclidExt}(5, 17) = 7$.

2.6 Stream ciphers

So far, we have illustrated cryptosystems that "reuse" the same key to encrypt letters or blocks of the plaintext. This is usually referred to as **block ciphers**.

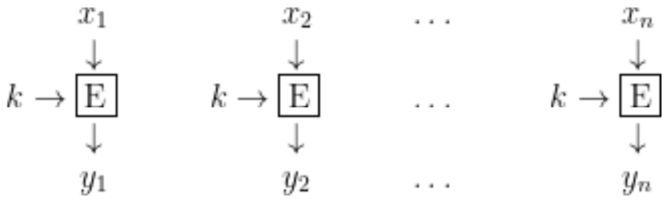


Figure 23: Block cipher

This scheme can be **generalized** by considering a **stream of keys** instead of a fixed one. Let $z_1, z_2, \ldots, z_n$ be such a stream. The idea is to **encrypt** the **first letter** of the plaintext with z_1, the **second** with z_2 and so on. It does not matter much if we encrypt a letter or a block, the important difference is that the used key is always different.
Having a **different key** for **each letter** or block of the plaintext is of course **appealing** but **not** much **practical**. The stream of key is thus usually derived starting from an

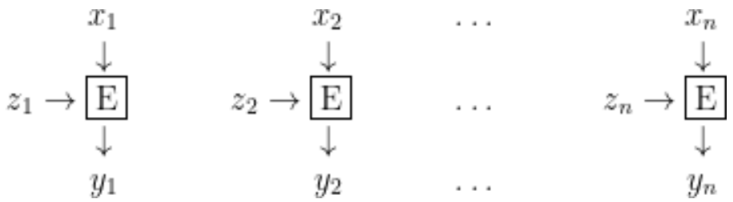


Figure 24: Cipher using different keys

initial key k. To make the **stream more complex** it can also **depend** on **previous part of the plaintext**. In general we say that

$$z_i = f_i(k, x_1, \dots, x_{i-1})$$

, i.e. the i-th key depends on k and on the previous $i-1$ letters (or blocks).
To understand why a key z_i cannot depend on plaintexts with indexes greater than or equal to i, it is useful to reason on the decryption scheme:

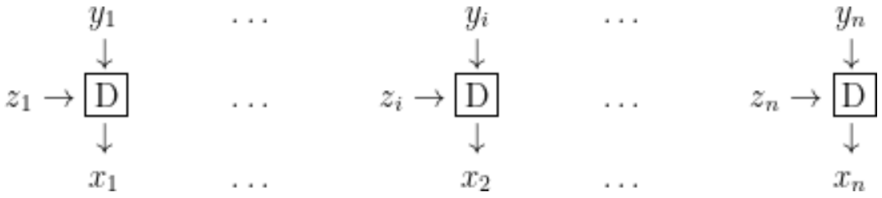


Figure 25: Decryption of a cipher using different keys

Now, $z_1 = f_1(k)$ meaning that we can compute it with not knowledge of the plaintext. To compute $z_2 = f_2(k, x_1)$, instead, we need to know x_1. As a consequence, we have to decrypt y_1 with z_1 before computing z_2. Once we have this key we can decrypt x_3 and compute z_3, and so on. The values are thus computed in the following sequence: $z_1, x_1, z_2, x_2, \dots, z_n, x_n$. It should be evident, now, that we cannot let z_i depend, e.g., on x_i as we would need that plaintext to compute the key.
Notice that **block ciphers** are, clearly, a **simple instance** of **stream ciphers** where $z_i = k$ for all i. It is also useful to classify these ciphers depending on certain properties of the key stream:

- **Periodic** stream ciphers;
- **Synchronous** stream ciphers;
- **Asynchronous** stream ciphers;

2.6.1 Periodic stream ciphers

A stream cipher is **periodic** if its key stream has the following form $z_1, z_2, \dots, z_d, z_1, z_2, \dots, z_d, z_1 \dots$, i.e., if it **repeats** after d steps.

Note that **Vigenére ciphers** can be seen as a **stream cipher** acting on **single letters** and with a **periodic key stream**. For example, if we want to formalize the cipher giving $(\mathcal{P}, \mathcal{C}, \mathcal{K}, E, D)$ and defining the key stream z_i, we have that:

- $E_{z_i}(x_i) = (x_i + z_i) \mod 26$;
- $D_{z_i}(y_i) = (y_i - z_i) \mod 26$;
- $z_i = k_i \mod m$.

2.6.2 Synchronous stream ciphers

A stream cipher is **synchronous** if its **key stream does not depend on plaintexts**, i.e., $z_i = f_i(k)$ for all i. When this happens, we have that the key stream can be generated starting from k and independently on the plaintext. This is particularly useful to improve efficiency: we do not need to obtain x_i to compute z_{i+1}. In fact, the key stream can be generated offline, before the actual ciphertext is received.
As an example, **Vigenére ciphers** can be seen as a **synchronous stream cipher**.

2.6.3 Asynchronous stream ciphers

This is the general case where $z_i = f_i(k, x_1, \ldots, x_{i-1})$. As mentioned above, we need to decrypt and compute the keys stream at the same time, as a key can depend on previous plaintexts.
We give a simple example of a cipher of this class, called **Autokey**. We let $\mathcal{P} = \mathcal{C} = \mathcal{K} = \mathbb{Z}_{26}$. $E_z(x) = x + z \mod 26$ and $D_z(y) = y - z \mod 26$, i.e., **encryption** and **decryption** are exactly as in a **shift cipher**. The key stream is defined as $z_1 = k$ and $z_i = x_{i-1}$ for $i \geq 2$, meaning that the first key in the stream is the initial key k while the next keys are the same as the previous plaintext.

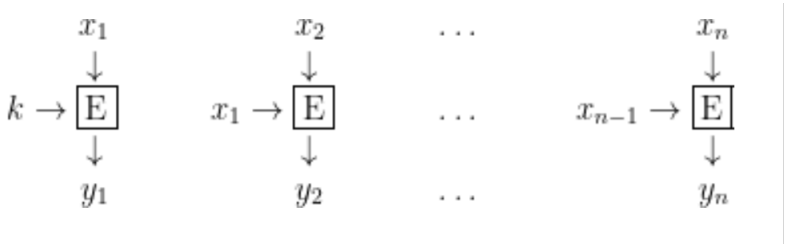


Figure 26: Autokey cipher

Example: consider the autokey cipher, and assume that the plaintext is the word "networksecurity", what is the encryption using $k = 5$ (recall: $E_z(x) = x + z \mod 26$)? The corresponding numbers are 13, 4, 19, 22, 14, 17, 10, 18, 4, 2, 20, 17, 8, 19, 24, so the encryption is $z_1 = 5$, $z_2 = x_1 = 13$, $z_3 = x_2 = 4$ etc.., so the result numbers are 18, 17, 23, 15, 10, 5, 1, 2, 22, 6, 22, 11, 25, 1, 17, and the corresponding ciphertext is SRXPKFBCWGWLZBR. For the encryption, we have that $D_z(y) = y - z \mod 26$, so we can use $k = 5$ to find the first letter x_1 of the plaintext: $x_1 = (18 - 5) \mod 26 = 13$. Then, we can use x_1 as a key to find x_2 : $x_2 = (17 - 13) \mod 26 = 4$ etc..
Notice that the **autokey** cipher is **insecure**, since there are only 26 different keys.

2.7 Perfect ciphers

We now discuss a **theoretical result** on the security of cryptosystems. We ask whether **perfect ciphers** exists, i.e., ciphers that can never be broken, even with after an unlimited time (informal definition). Interestingly, we will see that these ideal ciphers **exist** and **can be implemented in practice** but they are, in fact, **unpractical**. The theory, developed by **Claude Shannon**, assumes an **only-ciphertext** model of the **attacker**, i.e., the attacker only knows the ciphertext y and tries to find plaintext x or key k.
Another informal definition of perfect cipher is the following: a cipher system is said to offer perfect secrecy if, on **seeing** the **ciphertext** the interceptor gets **no extra information** about the **plaintext** than he had before the ciphertext was observed. In a cipher system with perfect secrecy the interceptor is "forced" to guess the plaintext.

2.7.1 Probability distribution

We call:

- $p_{\mathcal{P}}(x)$ the **probability** of a **plaintext** x to occur;
- $p_{\mathcal{K}}(k)$ the **probability** of a certain **key** k to be used as encryption key.

These two probability distributions induce a **probability distribution** on the **ciphertexts**. In fact, given a plaintext and a key there exists a unique corresponding ciphertext. We can compute such a probability distribution as follows:

$$p_{\mathcal{C}}(y) = \sum_{k \in \mathcal{K}, \exists x. E_k(x) = y} p_{\mathcal{K}}(k) p_{\mathcal{P}}(D_k(y))$$

Given a **ciphertext** y we look for **all the keys** that **can give** such a **ciphertext** from some **plaintext** x. We then sum the probability of all such keys times the probability of the corresponding plaintext.
Example: consider the following toy-cipher with $P = \{a, b\}$, $K = \{k_1, k_2\}$, $C = \{1, 2, 3\}$. The encryption is defined by the following table:

E	a	b
k_1	1	2
k_2	2	3

Table 1: Caption

We now let $p_p(a) = 3/4$, $p_p(b) = 1/4$, $p_k(k_1) = p_k(k_2) = 1/2$. Let us now compute $p_c(1)$:

$$p_c(y) = \sum_{k \in \mathcal{K}, \exists x. E_k(x) = y} p_k(k) p_p(D_k(y))$$

Thus, $p_c(1) = p_k(k_1) p_p(D_k(1)) = 1/2 * 3/4 = 3/8$.

2.7.2 Conditional probability

We can also compute the **conditional probability** of a **ciphertext** y with respect to a **plaintext** x. This gives a measure of how likely is a certain ciphertext once we fix a plaintext.

$$p_{\mathcal{C}}(y|x) = \sum_{k \in \mathcal{K}, E_k(x)=y} p_{\mathcal{K}}(k)$$

It is simply the sum of the probability of all keys giving y from x.
Example: the conditional probability of ciphertext 1 w.r.t. the two plaintexts a and b is:

$$p_{\mathcal{C}}(1|a) = \sum_{k \in \mathcal{K}, E_k(x)=y} p_{\mathcal{K}}(k) = p_{\mathcal{K}}(k1) = 1/2$$

$$p_{\mathcal{C}}(1|b) = \sum_{k \in \mathcal{K}, E_k(x)=y} p_{\mathcal{K}}(k) = 0$$

Notice, in particular, that 1 can never be obtained from b.
Once we have these values, the idea is to compute the **conditional probability** of a **plaintext** with respect to a **ciphertext**. This is very related to the **security** of the cipher, since it is a measure of how likely is a plaintext once a ciphertext is observed (which is what the attacker is usually interested to know). Interestingly, this conditional probability can be computed through that **Bayes theorem**:

$$p_{\mathcal{P}}(x|y) = \frac{p_{\mathcal{P}}(x)p_{\mathcal{C}}(y|x)}{p_{\mathcal{C}}(y)}$$

This conditional probability is quite useful when we'll prove that a cipher is perfect if $\forall x, y,\ p_{\mathcal{P}}(x|y) = p_{\mathcal{P}}(x)$.
Example: we can now compute the probabilities of plaintexts a and b with respect to ciphertext 1. We obtain:

$$p_{\mathcal{P}}(a|1) = \frac{p_{\mathcal{P}}(a)p_{\mathcal{C}}(1|a)}{p_{\mathcal{C}}(1)} = \frac{3/4 \times 1/2}{3/8} = 1$$

$$p_{\mathcal{P}}(b|1) = \frac{1/4 \times 0}{3/8} = 0$$

Thus, when observing 1 we are sure it is plaintext a and that it's not plaintext b, meaning that this cipher is completely insecure. The same happen if we compute the probabilities of a and b when observing 3 (in this case we're sure that it is plaintext b and not a, i.e. $p_{\mathcal{P}}(b|3) = 1$ and $p_{\mathcal{P}}(a|3) = 0$). For ciphertext 2 we have an interesting, less extreme, situation. We obtain, in fact, $p_{\mathcal{P}}(a|2) = 3/4$ and $p_{\mathcal{P}}(b|2) = 1/4$, thus a is more likely than b when 2 is observed. This might suggest that even for this ciphertext the attacker gains information (even if partial) about the plaintext. However, it is important to notice that $p_{\mathcal{P}}(a) = 3/4, p_{\mathcal{P}}(b) = 1/4$, i.e., that the probability of the two plaintexts, when 2 is observed, is exactly the one they occur in a message. In fact, observing 3 does not change anything.

2.7.3 Formal definition

We now give the definition of perfect cipher:
Definition (Perfect cipher). A cipher is **perfect** if and only if $p_{\mathcal{P}}(x|y) = p_{\mathcal{P}}(x)$ for all $x \in \mathcal{P}$ and $y \in \mathcal{C}$.
Intuitively, a cipher is perfect if **observing** a **ciphertext** y gives **no information** about any of the possible **plaintexts** x.
The cipher in the example is far from being perfect, but it satisfies the above definition for ciphertext 2. Concerning ciphertext 3, we have that $p_{\mathcal{P}}(a) = 3/4 \neq p_{\mathcal{P}}(a|3) = 0$, and $p_{\mathcal{P}}(b) = 1/4 \neq p_{\mathcal{P}}(b|3) = 1$, so the property does not hold.
Another possible definition is the following: if we consider a completely **bipartite graph** composed by **plaintexts** and **ciphertexts**, as the one in Picture 2.7.3, a cipher is defined as **perfect** if there is some **key** that **maps** any **message** to any **ciphertext** with **equal probability**. In this sense, the weights of the graph will all have the same weight equal to $1/n$.

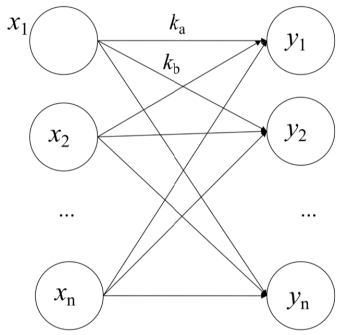


Figure 27: Completely bipartite graph

Exercise: prove that shift cipher with $p_{\mathcal{K}}(k) = \frac{1}{|\mathcal{K}|} = \frac{1}{26}$, i.e., with keys picked at random for each letter of the plaintext, is a perfect cipher. In other words, if we change key any time (not feasible in practice) and we encrypt a letter, then the shift cipher becomes perfect, i.e. unbreakable.
Solution: the idea for solving this exercise is to rely on the formal definition we gave above. By using the conditional probability, if we show that $p_{\mathcal{C}}(y|x) = p_{\mathcal{C}}(y)$ for every x and y, then $p_{\mathcal{P}}(x|y) = p_{\mathcal{P}}(x)$ for every x and y, so the cipher is perfect.
We recall that a shift cipher is defined as:

- $\mathcal{P} = \mathcal{C} = \mathcal{K} = \mathbb{Z}_{26}$;
- $E_k(x) = (x + k) \mod 26$;
- $D_k(y) = (y - k) \mod 26$.

We compute the probability of a generic ciphertext y as:

$$\begin{aligned} p_{\mathcal{C}}(y) &= \sum_{k\in\mathcal{K},\exists x.E_k(x)=y} p_{\mathcal{K}}(k)p_{\mathcal{P}}(D_k(y)) \\ &= \frac{1}{26}\sum_{k\in\mathcal{K},\exists x.E_k(x)=y} p_{\mathcal{P}}(D_k(y)) \\ &= \frac{1}{26}\sum_{k\in\mathcal{K}} p_{\mathcal{P}}(y-k \mod 26) \\ &= \frac{1}{26}\sum_{x\in\mathcal{P}} p_{\mathcal{P}}(x) = \frac{1}{26} \end{aligned}$$

Notice that:

- The first step comes from the fact that each $p_{\mathcal{K}}(k)$ is independent of we key we choose, since it is a constant value equal to $\frac{1}{26}$;
- The last two steps hold since for each key k, we always have a plaintext that gives y when encrypted under k. This plaintext is exactly $y-k \mod 26$. So the constraint $\exists x.E_k(x) = y$ always holds and $D_k(y) = y-k \mod 26$;
- Then, it is sufficient to observe that $y-k \mod 26$ for all possible keys gives exactly the set of all possible plaintexts $\mathcal{P}$ and the sum of all their probabilities gives 1.

We can now compute

$$p_{\mathcal{C}}(y|x) = \sum_{k\in\mathcal{K},E_k(x)=y} p_{\mathcal{K}}(k) = p_{\mathcal{K}}(y-x \mod 26) = \frac{1}{26}$$

Here it is enough to observe that, given x and y, there exists a unique key that encrypts x as y, which is precisely $y-x \mod 26$ (derived from $y = (x+k) \mod 26$).
Now Bayes theorem gives:

$$p_{\mathcal{P}}(x|y) = \frac{p_{\mathcal{P}}(x)p_{\mathcal{C}}(y|x)}{p_{\mathcal{C}}(y)} = \frac{p_{\mathcal{P}}(x)\frac{1}{26}}{\frac{1}{26}} = p_{\mathcal{P}}(x)$$

which gives the thesis.
We have seen that if we change key any time we encrypt a letter, a cipher as simple as the shift cipher becomes perfect, i.e., unbreakable. We now present two general results that, in fact, show that this strong requirement is indeed necessary and we cannot hope to develop perfect ciphers without it.

2.7.4 Important theorems

Theorem 1. Let $p_{\mathcal{C}}(y) > 0$ for all y. A cipher is perfect **only if** $|\mathcal{K}| \geq |\mathcal{P}|$.
The theorem states a **necessary condition** of a cipher to be perfect: it must be that the number of keys is at least the same as the number of plaintexts. In other words:

$$\text{perfect cipher } \rightarrow |\mathcal{K}| \geq |\mathcal{P}|$$

and, conversely,

$$|K| < |P| \rightarrow \text{ not perfect cipher}$$

Thus, besides the formal definition, we have a very easy way of proving that the cipher is not perfect

Proof: we first notice that by Bayes theorem we have that a cipher is perfect if an only if $p_{\mathcal{C}}(y|x) = p_{\mathcal{C}}(y)$ for all $x \in \mathcal{P}$ and $y \in \mathcal{C}$. If we fix x we obtain that for each y, $p_{\mathcal{C}}(y|x) = p_{\mathcal{C}}(y) > 0$ meaning that there exists at least one key k such that $E_k(x) = y$ (otherwise we would have $p_{\mathcal{C}}(y|x) = 0$). Notice also that all such keys are different since E_k is a function and we have fixed x. In fact, x cannot be mapped to two different ciphertexts by the same key (otherwise E_k would not be a function). Thus we have at least one key for each ciphertext meaning that $|\mathcal{K}| \geq |\mathcal{C}|$. Since, for any cipher, E_k injects the set of plaintexts into the set of ciphertext, we also have $|\mathcal{C}| \geq |\mathcal{P}|$, which gives the thesis $|\mathcal{K}| \geq |\mathcal{P}|$.

Theorem 2. Let $|\mathcal{P}| = |\mathcal{C}| = |\mathcal{K}|$. A cipher is perfect **if and only if**:

1. $p_{\mathcal{K}}(k) = \frac{1}{|\mathcal{K}|} \quad \forall k \in \mathcal{K}$;
2. For each $x \in \mathcal{P}$ and $y \in \mathcal{C}$ there exists exactly one key k such that $E_k(x) = y$.

Intuitively, the theorem states that for a cipher to be **perfect** (under the hypothesis that the size of the set of plaintexts, ciphertexts and key is the same) **keys** should be **picked at random** for any **encryption** and each plaintext is mapped into each ciphertext through a unique key.

Conversely, in order to show that a cipher is not perfect, we need to show that either condition (1) or (2) does not hold.

Proof: we prove that a perfect cipher implies the two above conditions. We leave the other side of the implication as an exercise. In Theorem 1 we have seen that, for perfect ciphers, if we fix x we obtain that for each y that $p_{\mathcal{C}}(y|x) = p_{\mathcal{C}}(y) > 0$ meaning that there exists at least one key k such that $E_k(x) = y$ and all of these keys are different. Thus we have $|\mathcal{K}| \geq |\mathcal{C}|$. In this theorem we have assumed $|\mathcal{K}| = |\mathcal{C}|$, meaning that all of these keys k are unique (otherwise we would have $|\mathcal{K}| > |\mathcal{C}|$). Since this holds for each x and y we have proved condition 2. To prove condition 1, it is enough to notice that $p_{\mathcal{C}}(y|x) = p_{\mathcal{K}}(k)$, i.e., the probability of y given x is equal to the probability of the unique key k that encrypts x into y. Thus, $p_{\mathcal{K}}(k) = p_{\mathcal{C}}(y|x) = p_{\mathcal{C}}(y)$ (only one key k maps x into y). If we fix y and we consider all possible plaintexts x we obtain all possible keys k and for all of them it holds $p_{\mathcal{K}}(k) = p_{\mathcal{C}}(y)$, with $p_{\mathcal{C}}(y)$ constant. Given that the sum of the probability of all keys must be 1, we obtain $p_{\mathcal{K}}(k) = \frac{1}{|\mathcal{K}|}$ which proves condition 1.

2.7.5 The one-time-pad

We conclude giving a famous **example of a perfect cipher** that has been used in practice. This cipher has been used for the telegraph and is a binary variant of Vigenére with keys picked at random. More precisely we have $\mathcal{P} = \mathcal{C} = \mathcal{K} = \mathbb{Z}_2^d$ (i.e. plaintext, ciphertext and keys can be either 0 or 1, i.e. binary) with $p_{\mathcal{K}}(k) = \frac{1}{|\mathcal{K}|} = \frac{1}{2^d}$ for all $k \in \mathcal{K}$. Encryption is defined as $E_{(k_1,\ldots,k_d)}(x_1, \ldots, x_d) = (x_1 \oplus k_1, \ldots, x_d \oplus k_d)$ where $\oplus$ is the bitwise XOR operation. We recall that the XOR operation is defined as follows:

$$101 \text{ XOR } 011 = 110$$

We notice that the premise of Theorem 2 holds (set sizes are the same by definition of the cipher). Also condition 1 holds, i.e., $p_{\mathcal{K}}(k) = \frac{1}{|\mathcal{K}|}$ by definition of the cipher. We only

need to prove condition 2. Let $x \in \mathcal{P}$ and $y \in \mathcal{C}$. We have that the unique key giving y from x is computed as $x \oplus y$, i.e., $(x_1 \oplus y_1, \ldots, x_d \oplus y_d)$. We thus conclude that the cipher is perfect.
Despite being a perfect cipher, we notice how one-time-pad is pretty unfeasible in practice, since it needs to change the key each time.

2.7.6 Recap

Shannon theory on perfect ciphers shows that such ideal ciphers exist but require as many keys as the possible plaintexts, and keys need to be picked at random for each encryption. Even if this makes such ciphers unpractical, the one-time-pad has been used for real transmission. The setup consisted of two identical books with thousands of "random" keys. Each key was used once (from which the name one-time). Once the book had been used completely, new shared books were necessary.

2.8 Exercises

1. Decrypt the following ciphertext using substitution cipher (the language is English):

   GNDO DO L ODEFYK KRLEFYK CA L HDFNKIGKRG.

   XKYY BCWK!

2. Decrypt the following ciphertext using Vigenére cipher (the key is "FLUTE"): STWXXWJ;
3. Write a program that decrypts the following ciphertext (Vigenére cipher):

```
WTTNRAVWSKAMFFEVREBKZXMKLCLANMOZSWDXKOHKTQDMCDVWOIIUHXZWXIYVNRYSWNUZAFCVZEUMIKKU
XSGFYMZCGADGVYBIZSZXPPCIYMEZIXZWXWZVUIMWZYNIJOWACPNHDRXHOXALDIOYBTDGKALIKGNKZYII
MWRZCSFKEVTENRYAYGNMKLDMFDRSAESUIOLZKCRSGEGMTKQMLZKCEAVGOYIKMSJUKVGPEWVXZUHWGKDM
HLFGJRJOXIJAJOTYUIMSNTGMFVWAHPYPVWJNYGGMHLQZEXCIYUNHSQIIOPUIMJVKFZWJUAWPRTTEEJMX
MHELHSRXCIIZBIHDAIDFEIJJDGEMFYWLLTFCXLROHAGMHLMPJSMXYZBILJKCMWRSAKVZNMFYQXLYQTDG
BOHKLZALLTFMZWNNYRKMZLPYYCVAYONIJSXJTEJMOLGOHOWQAACLAGGSJKVCZWNBSPEIREJTIXZAJODZ
IIMCIKGEJCWJWPVROLRZHSJVELLWVGZXYOHOALOWGPECHYTNIYLGBBSPJETXFNYEJLDMCLOFDXJGSXGA
PAPWSSCHVGLSZVAICTFLVPCHYPSLAESPAWCIKNIYYZPQEZIIMEWZYVOSNLDTGSXGLXLIVLKPPCGLVXJN
YSMYDBEZUEQINUHHWJALLEGLDWSANELLDMETZIDXRRFWWWIMOBHMOIEGNYJSHJFEJLZRKNYVSTXQELPX
PECRSXGGGIHLGGCSLZIJALOELTFXXSRZJSUCABLYQPJSBKXELAPIYOGLZRYALVAWZWYLYMXIJZUVLWZB
ZSRVAIVZZSJAPNWLFLZHRILSKKDMCXVRYXYGNWZWDIOYRZZVSKZSJWOMPYNVVFSONAALDMTEUIMENGCW
LUKIEABGFIKULEOSPKSEBXVOVUOXGXEBLYQFPVEOHKOAPPNFEMJWZZSWZWNIYLPVJWJZBIXAATOLSXZV
ZZURVXKZEFAEOICEQEKBQAETAXDQVZIWWWEBAZCHJAEGFEJYAZLMOMOLFRYYFVAZESRLZHXK
```

4. Encrypt and decrypt message $(2, 5)$ using the Hill cipher with $K = \begin{bmatrix} 5 & 11 \\ 8 & 3 \end{bmatrix}$;
5. Encrypt and decrypt message $(1, 3)$ using the Hill cipher with $K = \begin{bmatrix} 1 & 2 \\ 4 & 3 \end{bmatrix}$;
6. Encrypt and decrypt message $(3, 2)$ using the Hill cipher with $K = \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}$

   Solution on slide 2 of L11;
7. Suppose that we know that FRIDAY has been encrypted as PQCFKU using the Hill cipher, so we have (FRIDAY, PQCFKU). What is the key K? Assume K is a 2x2 matrix. Solution on slide 29 of L5;
8. Compute $\gcd(17, 2)$;
9. Suppose $X = \begin{bmatrix} 5 & 17 \\ 8 & 3 \end{bmatrix}$ and $Y = \begin{bmatrix} 15 & 16 \\ 2 & 5 \end{bmatrix}$. We know that if X^{-1} exists, then $K = X^{-1}Y \mod 26$. Compute X^{-1} using the extended euclidean algorithm and retrieve K. Solution on slides 34-35-36 of L5;
10. Compute $\text{EuclidExt}(14, 17)$;
11. Compute $\text{EuclidExt}(17, 5)$. Solution on slide 38 of L5;
12. Decrypt the following ciphertext, which was encrypted using a *shift cipher*: BEEAKFYDJXUQYHYJIQRYHTYJIQFBQDUJIIKFUHCQD. Solution on slide 27 of L5;

13. Try to break the following ciphertext encrypted with the autokey cipher: GUAAMLXOOVTMRVTKXOWSSDXNVJSTVTACALTNQFTPNIHUXRPWLV;
14. Try to extract the plaintext from this word encoded using the autokey cipher. Notice that we do not know the key k: FTPNIH. Solution on slide 31-32 of L6;
15. Prove that the cipher with the following encryption function $E_k(x_1, .., x_d) = (x_1 + k, .., x_d + k) \mod 26$ is not perfect. Use both the formal definition and the theorem 1 discussed in class. Solution on slide 3-4 of L7;
16. Consider the following cipher with:

    - $P = \{a, b\}$;
    - $K = \{k_1, k_2, k_3\}$;
    - $C = \{1, 2, 3, 4\}$;
    - $E_{k_1}(a) = 1$, $E_{k_2}(a) = 2$, $E_{k_3}(a) = 3$, $E_{k_1}(b) = 2$, $E_{k_2}(b) = 3$, $E_{k_3}(c) = 4$

    We now let $p_p(a) = 1/4$, $p_p(b) = 3/4$, $p_k(k_1) = 1/2$, $p_k(k_2) = p_k(k_3)1/4$. Compute $p_p(a|1)$, $p_p(a|2)$, $p_p(a|3)$, $p_p(a|4)$, $p_p(b|1)$, $p_p(b|2)$, $p_p(b|3)$, $p_p(b|4)$.

3 Modern cryptography

3.1 Composition of ciphers

So far, we have seen simple historical ciphers and we have discussed how to break them. **Modern ciphers**, however, are based on very **simple operations**, such as substitution, XOR, ..., that are **combined** in a smart way so to make the overall algorithm strong and really hard to analyse. It is important to keep in mind that **combining simple ciphers** does not **always improve security**.
For example, consider the shift cipher composed twice. We first shift by k_1 and then by k_2 modulo 26, as represented in Picture 3.1.

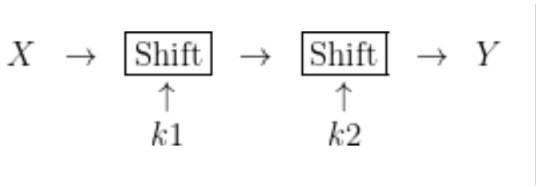


Figure 28: Composition of two shift ciphers - 1

It is clear that this is equivalent to shifting by $k_1 + k_2$ modulo 26, meaning that applying twice the cipher is the same as applying it once with a key given by the sum of the two keys.

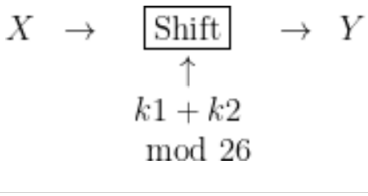


Figure 29: Composition of two shift ciphers - 2

This informal reasoning can be made more precise.
Definition (Composition). We consider two ciphers $S^1 = (\mathcal{P}^1, \mathcal{C}^1, \mathcal{K}^1, E^1, D^1)$ and $S^2 = (\mathcal{P}^2, \mathcal{C}^2, \mathcal{K}^2, E^2, D^2)$. We let $\mathcal{P}^1 = \mathcal{C}^1 = \mathcal{P}^2 = \mathcal{C}^2$, that we note as $\mathcal{P}$ and $\mathcal{C}$ in the following. In this way the output of one cipher is for sure a possible plaintext for the second cipher. We can now define **composition** as $S^1 \times S^2 = (\mathcal{P}, \mathcal{C}, \mathcal{K}^1 \times \mathcal{K}^2, E, D)$ with

$$E_{(k1,k2)}(x) = E^2_{k2}(E^1_{k1}(x))$$

$$D_{(k1,k2)}(y) = D^1_{k1}(D^2_{k2}(y))$$

As we can see, in order to encrypt a plaintext x using a composition of two ciphers with keys $k1$ and $k2$ we must:

1. **Encrypt** x using the **first encryption function** (using $k1$);
2. **Encrypt** the **result** of (1) using the **second encryption function** (using $k2$).

Example: consider the composition of the two shifts above. Formally we have that $\overline{E_k^1(x) = E_k^2(x) = x + k \mod 26}$. Thus,

$$\begin{aligned} E_{(k1,k2)}(x) &= E_{k2}^2(E_{k1}^1(x)) = (x + k1 \mod 26) + k2 \mod 26 \\ &= x + (k1 + k2 \mod 26) \mod 26 = E_{k1+k2 \mod 26}^1(x) \end{aligned} \tag{1}$$

This proves that composing the shift cipher twice is equivalent to applying it once using as a key the sum of the two keys $k1$ and $k2$, modulo 26.

3.1.1 Idempotent ciphers

We have seen that the shift cipher, when repeated twice is equivalent to itself with a different key. When this happens, the cipher S is said to be **idempotent**, written $S \times S = S$. In this case we know that iterating the cipher will be of no use to improve its security. Even if we repeat it n times we will still get the initial cipher, i.e., $S^n = S$.
We have mentioned that modern ciphers are based on simple operations composed together. Another ingredient is, in fact, **iteration**. Almost any modern cipher repeats a **basic core of operations** for a certain number of **rounds**. It is thus necessary that such core operations do not constitute an idempotent cipher.
It can be proved that if we have two **idempotent ciphers** that commute, i.e., such that $S^1 \times S^2 = S^2 \times S^1$, then their **composition** is also **idempotent**. In this case, we know that iterating their composition is useless. To see why this holds consider one iteration of their composition (recall that function composition is associative):

$$\begin{aligned} &(S^1 \times S^2) \times (S^1 \times S^2) && \\ &= S^1 \times (S^2 \times S^1) \times S^2 && \text{associative property} \\ &= S^1 \times (S^1 \times S^2) \times S^2 && \text{commutative property} \\ &= (S^1 \times S^1) \times (S^2 \times S^2) && \text{associative property} \\ &= S^1 \times S^2 && \text{idempotence of the initial ciphers} \end{aligned}$$

3.1.2 Recap

We have seen examples of how algebraic properties, such as commutativity, can help simplifying the analysis of a cipher. When developing a robust cipher we need to avoid as much as possible that operations can be rearranged, swapped, simplified.

3.2 The AES cipher

The **Advanced Encryption Standard** (AES) has been selected by the National Institute of Standards and Technology (NIST) after a five-year long competition. The original name of the cipher is Rijndael from the names of the two inventors, the cryptographers Joan Daemen and Vincent Rijmen. As any modern cipher, AES is the **composition** of rather **simple operations** and contains a **non-linear component** to **avoid known-plaintexts attacks** (as the one we have seen on the Hill cipher). The composed operations give a **non-idempotent cipher** that is **iterated** for a fixed number of **rounds** (the longer the key, the more rounds are executed).
Rijndael has been selected because it resulted to be the best one providing:

- High **security** guarantees;
- High **performance**;
- **Flexibility** (different key length).

All of these **features** are, in fact, **crucial** for any modern cipher. Its predecessor, the Data Encryption Standard (DES) is still in use after almost 40 years, in a variant called Triple DES (3DES), which aims at improving the key length. In fact, DES key of only 56 bits is too short to resist brute-forcing on modern, parallel computers.

3.2.1 Mathematical background

AES works on the **Galois Field** with 2^8 elements noted $\mathbf{GF}(2^8)$. Intuitively, it is the **set** of all **8-bit digits** with **sum** and **multiplications** performed by interpreting the bits as (binary) **coefficients of polinomials**. For example, element 11010011 can be seens as $x^7 + x^6 + x^4 + x + 1$ while 00111010 is $x^5 + x^4 + x^3 + x$. The sum will thus be $x^7 + x^6 + x^4 + x + 1 + x^5 + x^4 + x^3 + x = x^7 + x^6 + x^5 + x^3 + 1$, since two 1's coefficient becomes 0, modulo 2, and the term disappears (for example $x + x = 2x = 0x = 0$). We see that **sum** and **subtraction** are just the **bit-wise XOR** of the binary numbers, i.e., $11010011 \oplus 00111010 = 11101001$ which is $x^7 + x^6 + x^5 + x^3 + 1$.
Product is done **modulo** the **irreducible polinomial** $x^8 + x^4 + x^3 + x + 1$. Irreducible means that it cannot be written as the product of two other polinomials (it is, intuitively, the equivalent of primality).
For example, $(x^7 + x^6 + x^4 + x + 1) \times (x^5 + x^4 + x^3 + x)$ gives

$$x^{12}+x^{11}+x^9+x^6+x^5+x^{11}+x^{10}+x^8+x^5+x^4+x^{10}+x^9+x^7+x^4+x^3+x^8+x^7+x^5+x^2+x$$

, which is reduced to

$$x^{12} + x^6 + x^5 + x^3 + x^2 + x$$

Now, the next step is to **divide** $x^{12}+x^6+x^5+x^3+x^2+x$ by the **irreducible polynomial** $x^8 + x^4 + x^3 + x + 1$, and find the remainder. In general, long division of polynomials is similar to long division of whole numbers, and when we divide two polynomials we can check the answer using:

$$\text{dividend} = (\text{quotient} \cdot \text{divisor}) + \text{remainder}$$

or, equivalently,

$$\text{dividend}/\text{divisor} = \text{quotient} + \frac{\text{remainder}}{\text{divisor}}$$

In our example, dividing $x^{12} + x^6 + x^5 + x^3 + x^2 + x$ by $x^8 + x^4 + x^3 + x + 1$ and finding the remainder results as follows:

1. Since $x^{12}/x^8 = x^4$, we shift 4 bits to the left the divisor;
2. Then, we perform a XOR operation between the dividend and the divisor;

3. Since the results contains 9 bits, which is too much, we perform another XOR operation between the previous result and the divisor;
4. Finally, we retrieve the remainder, which results to be 11000101, i.e., $x^7+x^6+x^2+1$.

1	0	0	0	0	0	1	1	0	1	1	1	0
1	0	0	0	1	1	0	1	1				
				1	1	1	0	1	1	1	1	0
				1	0	0	0	1	1	0	1	1
					1	1	0	0	0	1	0	1

This operation is **quadratic** in general, with respect to the number of bits (8). It can be optimized with the following linear algorithm (which is, in fact, a working python code):

```
def AESmult(a, b):
    p = 0                       # p is 0 at the beginning
    for i in range(0,8):        # for the 8 bits of a and b do:
        if b & 1 != 0:          # the least significant bit of b is set
            p = p ^ a           # sum a to p (xor)
        b >>= 1                 # shifts b to the right
        hbit = (a & 0x80) != 0  # true if the most significant bit of a is set
        a <<= 1                 # shifts a to the left
        if hbit:                # if the most significant bit of a was set
            a = a ^ 0x11b       # sum 100011011 to a (xor), this always returns
                                # a 8-bit number
    return p
```

Figure 30: Product - optimization

The **optimization** works as follows:

1. We let a to be the binary representation of the coefficients of the first term of the multiplication, and b the binary representation of the coefficients of the second one;
2. We let $p = 00$;
3. We perform a XOR operation between a and p, and we update p with the result of this operation;
4. Then, b is shifted to the right and a is shifted to the left meaning that we respectively divide and multiply by x the two polynomials. Now we erase the least significant bit of b and we add a 0 to a:
    - If the least significant bit is a 1, we continue performing the XOR operation between the new a and p;
    - Otherwise, we skip the XOR operation.
5. When a becomes more than 2^8 we need to XOR to it the modulus 100011011, i.e. $0x11b$, to keep it 8-bits long.

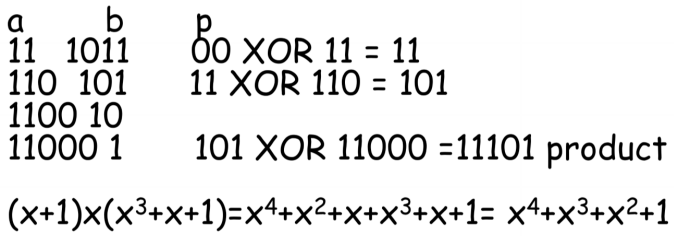


Figure 31: Product - optimization: example

An example is provided in Picture 3.2.1.
The **correctness** of this algorithm derives from the **invariant** which states that after each loop:

- $ab + p$ is the product of the initial a and b (all operation does in the Galois Field);
- Since b is 0 at the end, we have that p will contain the product.

3.2.2 The AES cipher

Now that we have introduced the basic operation used to implement AES we can describe the cipher. The official **description** of **AES** is available on-line.
AES operates on a **4×4 matrix** of bytes. We have that 16 bytes are 128 bits which is, in fact, the block size. **Plaintext** bytes $b_1, \ldots, b_{16}$ are copied in the matrix by columns following this scheme:

$$\begin{bmatrix} b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \\ b_4 & b_8 & b_{12} & b_{16} \end{bmatrix}$$

Cipher **keys** have lengths of 128, 192, and 256 bits. AES has **10 rounds** for 128-bit keys, **12 rounds** for 192-bit keys, and **14 rounds** for 256-bit keys. Rijndael was designed to handle additional block sizes and key lengths, however they are not adopted in the AES standard. A round is composed of different operations, all of which are invertible:

1. **AddRoundKey**: the round **key** (see Key Expansion, below) is bitwise **XOR**-ed with the **block**. A round key is thus 128 bits, independently of the chosen key size.

   In this example, $b_{2,2} = a_{2,2}$ XOR $k_{2,2}$.

2. **SubBytes**: a **fixed non-linear substitution**, called **S-box**, is applied to each byte of the block. The substitution is reported below. Given a **byte** in hexadecimal notation, the **first digit** is used to select a **row** and the **second** one to select a **column**. For example, $0x25$ would be the third row (2) and the sixth column (5) giving $0x3f$.

   The S-box is represented below:

   Notice that S-box is secure because the attacker cannot easily retrieve $b_{2,2}$, since 2^{16} possible values exist. Moreover, this S-box has been obtained by taking, for each

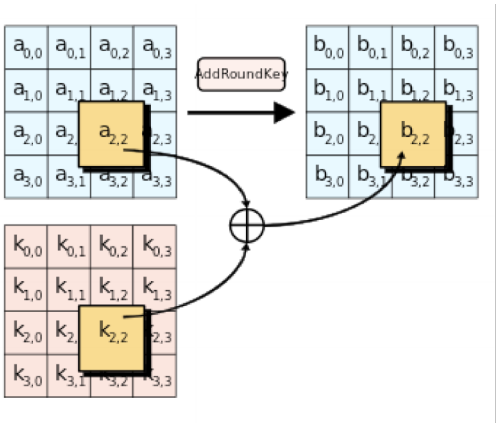


Figure 32: AddRoundKey operation

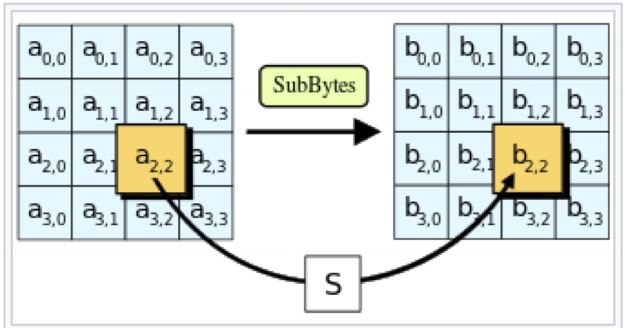


Figure 33: SubBytes operation

```
    0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
  ------------------------------------------------
0 |63 7c 77 7b f2 6b 6f c5 30 01 67 2b fe d7 ab 76
1 |ca 82 c9 7d fa 59 47 f0 ad d4 a2 af 9c a4 72 c0
2 |b7 fd 93 26 36 3f f7 cc 34 a5 e5 f1 71 d8 31 15
3 |04 c7 23 c3 18 96 05 9a 07 12 80 e2 eb 27 b2 75
4 |09 83 2c 1a 1b 6e 5a a0 52 3b d6 b3 29 e3 2f 84
5 |53 d1 00 ed 20 fc b1 5b 6a cb be 39 4a 4c 58 cf
6 |d0 ef aa fb 43 4d 33 85 45 f9 02 7f 50 3c 9f a8
7 |51 a3 40 8f 92 9d 38 f5 bc b6 da 21 10 ff f3 d2
8 |cd 0c 13 ec 5f 97 44 17 c4 a7 7e 3d 64 5d 19 73
9 |60 81 4f dc 22 2a 90 88 46 ee b8 14 de 5e 0b db
a |e0 32 3a 0a 49 06 24 5c c2 d3 ac 62 91 95 e4 79
b |e7 c8 37 6d 8d d5 4e a9 6c 56 f4 ea 65 7a ae 08
c |ba 78 25 2e 1c a6 b4 c6 e8 dd 74 1f 4b bd 8b 8a
d |70 3e b5 66 48 03 f6 0e 61 35 57 b9 86 c1 1d 9e
e |e1 f8 98 11 69 d9 8e 94 9b 1e 87 e9 ce 55 28 df
f |8c a1 89 0d bf e6 42 68 41 99 2d 0f b0 54 bb 16
```

Figure 34: S-box

byte, its **multiplicative inverse** in the finite field (that can be computed efficiently via an algorithm that we will see later on), noted $b_7, \ldots, b_0$, and applying the affine transformation $b_i = b_i \oplus b_{i+4 \mod 8} \oplus b_{i+5 \mod 8} \oplus b_{i+6 \mod 8} \oplus b_{i+7 \mod 8} \oplus c_i$, with c_i

representing the i-th bit of 01100011. The above transformation can be written as:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Figure 35: Generation of the S-box

Using multiplicative inverses is known to give **non-linear properties**, while the affine transformation complicates the attempt of algebraic reductions.

3. **ShiftRows**: **rows** of the block matrix are **shifted** to the left by 0,1,2,3, respectively. The shift is circular:

$$\begin{bmatrix} b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \\ b_4 & b_8 & b_{12} & b_{16} \end{bmatrix} \Rightarrow \begin{bmatrix} b_1 & b_5 & b_9 & b_{13} \\ b_6 & b_{10} & b_{14} & b_2 \\ b_{11} & b_{15} & b_3 & b_7 \\ b_{16} & b_4 & b_8 & b_{12} \end{bmatrix}$$

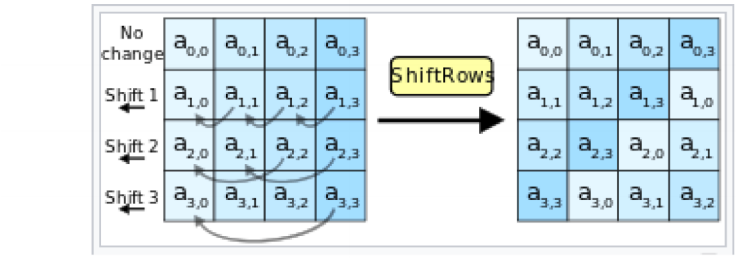


Figure 36: ShiftRows operation

4. **MixColumns**: columns of the block matrix are multiplied by the following matrix:

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

Figure 37: MixColumns operation

For example the first byte of each column is computed as $2c_0 \oplus 3c_1 \oplus c_2 \oplus c_3$.

NOTE: this fixed matrix is obtained by considering each column as a four-term polynomial with coefficients in $\mathbf{GF}(2^8)$. The columns are then multiplied modulo

$x^4 + 1$ with a fixed polynomial $a(x)$, given by $a(x) = 3x^3 + x^2 + x + 2$. This specific modulus is such that, e.g., x^4 becomes x^0, x^5 becomes x^1 and so on..

5. **Key Expansion** (not covered during the lecture): we have mentioned that AES uses round keys in the AddRoundKey step. These keys are in fact derived from the initial AES key as follows.

   Keys are represented as **arrays of words** of 4 bytes. So, for example, a 128 bit key will be 4 words of 4 bytes, i.e., 16 bytes. This is expanded into an array of size $4 * (N_r + 1)$, where N_r is the **number of rounds**. In this way we obtain 4 different words of key for each round.

   Let N_k note the **number of words** of the **initial key** (e.g. 4 for 128 bits). The first N_k words of the key array are the same as the initial key. Next i-th word is obtained from the previous $i - 1$ word, possibly transformed as described below, XOR-ed with word i-N_k. The transformation happens only for words in position multiple of N_k and consists of a cyclic left shift of word bytes by one position (*RotWord*) followed by a byte-wise application of the S-box (*SubWord*) and a XOR with a round constant (*Rcon*). This constant at step j is the word $[x^{j-1}, 0x00, 0x00, 0x00]$ with x^{j-1} computed in the Galois field, meaning 02^{j-1} since polynomial x is the binary number 00000010, i.e., $0x02$.

   The pseudocode for this phase is represented Picture 5.

```
for i in range(0,Nk):
    w[i] = k[i]    # copy the key words in the first
Nk words
for i in range(Nk, 4 * (Nr+1)):
    temp = w[i-1]
    if (i % Nk == 0):
        temp = SubWord(RotWord(temp)) ^ Rcon[i/Nk]
    w[i] = w[i-Nk] ^ temp
```

Figure 38: Key Expansion operation

   IMPORTANT NOTE: for 256 bit key, when i-4 is a multiple of N_k *SubWord* is applied to $w[i-1]$ before the XOR. This has been omitted in the code for the sake of readability. Moreover, note that of course the first byte of *Rcon* can be precomputed.

Here is the **overall scheme** for AES assuming that variable state is initialized with the 4x4 matrix of the plaintext (see above) and $w[]$ has been initialized by key expansion.
Notice that in the **last round we do not perform** the *MixColumn* **operation**.
Decryption is computed by **applying inverse operations**.
Notice that:

1. *AddRoundKey* is unchanged since XOR is the inverse of itself;
2. *InvShiftRows* trivially amounts to revert the shifts on the row (to the right instead of left);

```
AddRoundKey(state, w[0, 3])

for round in range(1,Nr):
    SubBytes(state)
    ShiftRows(state)
    MixColumns(state)
    AddRoundKey(state, w[round*4, round*4+3])

SubBytes(state)
ShiftRows(state)
AddRoundKey(state, w[Nr*4, Nr*4+3])
```

Figure 39: Scheme of AES - Encryption

```
AddRoundKey(state, w[Nr*4, Nr*4+3])

for round in range(Nr-1,0,-1):
    InvShiftRows(state)
    InvSubBytes(state)
    AddRoundKey(state, w[round*4, round*4+3])
    InvMixColumns(state)

InvShiftRows(state)
InvSubBytes(state)
AddRoundKey(state, w[0, 3])
```

Figure 40: Scheme of AES - Decryption

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figure 41: Inverse S-Box

3. *InvSubBytes* is computed by using the following inverse substitution of the S-Box:
4. Finally, *InvMixColumns* is given by the following operation:

   For example the first byte of each column is computed as $0ec_0 \oplus 0bc_1 \oplus 0dc_2 \oplus 09c_3$.

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

NOTE: this fixed matrix is obtained by considering each column as a four-term polynomial with coefficients in $\mathbf{GF}(2^8)$. The columns are then multiplied modulo x^4+1 with the inverse of the fixed polynomial $a(x)$, given by $a^{-1}(x) = 0bx^3+0dx^2+09x+0e$.

The algorithm for decryption is written in a form similar to the one for encryption but operations are not in the same order. It can, in fact, become the very same algorithm by noticing that:

1. *SubBytes* and *ShiftRows* commute. It does not matter if we first apply the byte-wise substitution or if we first shift the rows. The final result will be the same. Of course, the same holds for the inverse transformations;
2. $InvMixColumns(\text{state} \oplus \text{roundKey}) = InvMixColumns(state) \oplus InvMixColumns(roundKey)$. This allows for inverting the two functions, provided that *InvMixColumns* is applied to the all the round keys.

Call *dw* the array containing the round keys transformed via InvMixColumns. The final decryption algorithm is represented in Picture 3.2.2.

```
AddRoundKey(state, dw[Nr*4, Nr*4+3])

for round in range(Nr-1,0,-1):
    InvSubBytes(state)
    InvShiftRows(state)
    InvMixColumns(state)
    AddRoundKey(state, dw[round*4, round*4+3])

InvSubBytes(state)
InvShiftRows(state)
AddRoundKey(state, dw[0, 3])
```

Figure 42: Decryption algorithm

This is exactly the same as the one for encryption, but with the inverse functions. Having the same algorithm for encryption and decryption simplifies a lot implementations, especially if they are done in hardware.
The final scheme of the AES algorithm is provided in Picture 3.2.2: note that AES is a symmetric key algorithm.
Finally, we recall the fact that the security of AES depends on the number of rounds that are executed: the more rounds, the more secure the algorithm is.

3.3 Block cipher modes of operation

When using block ciphers we have to face the problem of encrypting **plaintexts** that are **longer** than the **block size**. We then adopt a **mode of operation**, i.e., a **scheme** that

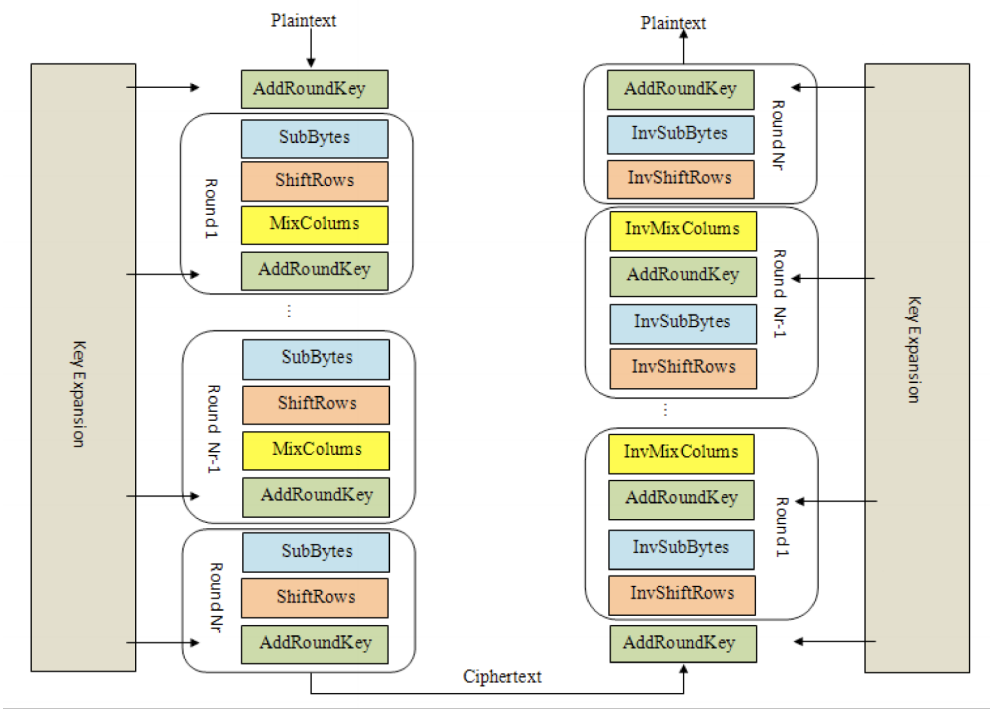


Figure 43: AES algorithm

repeatedly applies the **block cipher** and allows for encrypting a plaintext of arbitrary size.

3.3.1 Electronic CodeBlock mode (ECB)

This is the simplest mode and is, in fact, what we have done so far with classic ciphers: the **plaintext** X is split into **blocks** $x_1, x_2, \ldots, x_n$ whose **size** is exactly the same as the size of the **cipher block**. Each **block** is then **encrypted** independently using the fixed **key** k. For example, a substitution cipher applies to letters. What we do is to split the plaintext into single letters that are encrypted independently.

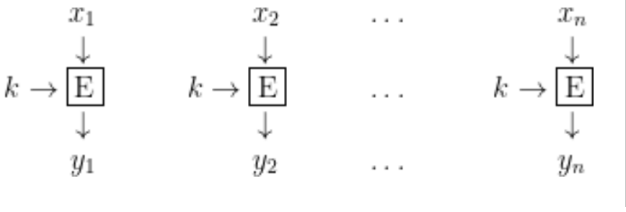


Figure 44: ECB: Encryption

Decryption is done, as expected, by reversing the scheme:
Advantage: this scheme has the advantage of being very **simple** and **fast**, especially on **multi-core computers**. Notice, in fact, that each single encryption/decryption can be performed **independently**.
Disadvantages: the **security** of the scheme, however, is **poor**. Indeed:

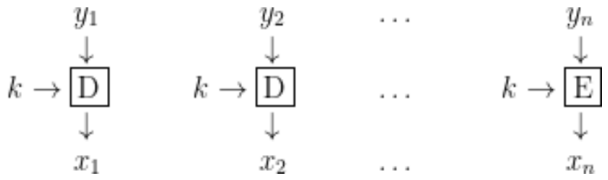


Figure 45: ECB: Decryption

1. It mainly conveys all the **defects** of **monoalphabetic classic ciphers**: equal plaintext blocks are encrypted in the same way. This allows for the construction of a code-book (from which the mode name) mapping ciphertexts back to plaintexts. It is often the case, in practice, that part of a plaintext is fixed due to the message format, for example. Think of a mail starting with "Dear Alice, ...". If we know a part of the plaintext, we know how the blocks containing that part are encrypted. We can use this information to decrypt other parts of the message, whenever we see the same block occurring.

   Picture 1 provides an immediate visualization of the codebook problem described above.

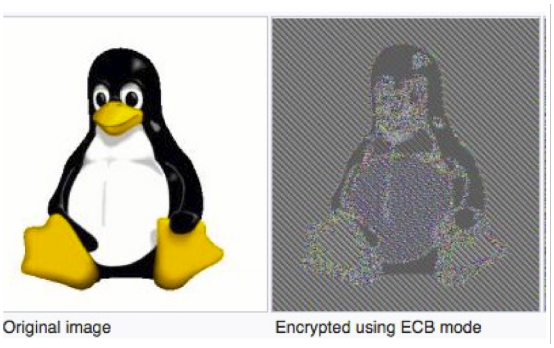


Figure 46: Attack on ECB

2. Another crucial limitation of this mode is the complete **absence of integrity**: an **attacker** in the middle might duplicate, swap, eliminate encrypted blocks and this would correspond to a plaintext where the same blocks are duplicated, swapped, eliminated. Again, having information about the format of the plaintext, an attacker might be able to obtain a different meaningful plaintext. How critical is this attack really depends on the application. But it is not a good idea to leave such an easy opportunity.

3.3.2 Cipher Block Chaining mode (CBC)

This mode solves or mitigates all the issues of ECB discussed above: it **prevents equal plaintexts** to be **encrypted** the **same way** and, at the same time, it provides a **higher degree of integrity**, even if it is not yet satisfactory on this aspect. The idea is to **"chain" encryption of blocks** using the **previous encrypted block**. The first block

is chained with a special number called *Initialization Vector* (IV) that is kept secret together with key k.

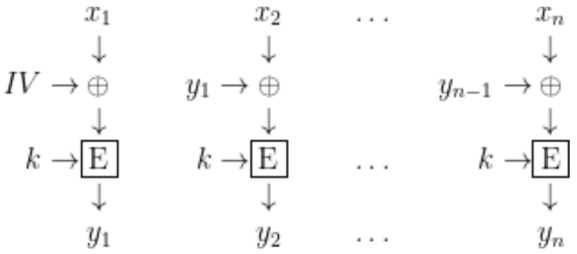


Figure 47: CBC: Encryption

Decryption is as follows:

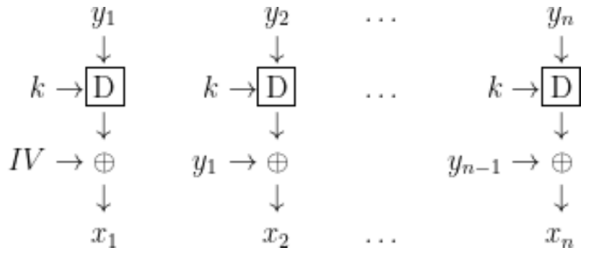


Figure 48: CBC: Decryption

Advantage: as mentioned above, **CBC never encrypts the same plaintext block in the same way**, preventing the code-book attack. **Integrity** is improved, but is not yet completely satisfactory. If an attacker swaps, duplicates or eliminates encrypted blocks this will result in at least one corrupted plaintext block. Notice however that this might be unnoticed at the application level and, again, we cannot leave to the application the whole task of checking integrity of decrypted messages.
Disadvantage: using **XOR** introduces a **new weakness**: the **attacker** manipulating **one bit** of an **encrypted block** y_i obtains that the **same bit of plaintext** x_{i+1} is also **manipulated**. At the same time x_i is corrupted.

3.3.3 Output FeedBack mode (OFB)

We now see two modes of operation that "transform" block ciphers into stream ciphers. The general idea is to use the block cipher to generate a complex key stream. Encryption is then performed by just XOR-ing the plaintext blocks with the keys of the stream. Intuitively, this is like one-time-pad with a generated key stream. The more the stream is close to a random stream the more the cipher will be close to a perfect one.
Notice that key generation is completely independent of the plaintext and ciphertext. In fact, it is possible to generate the key stream offline, having key k, and perform encryption later on, when necessary. Decryption simply consists of "swapping the arrows"

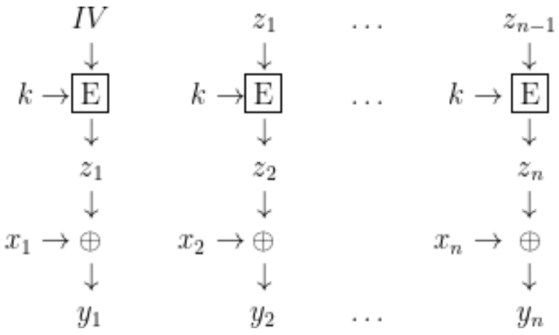


Figure 49: OFB: Encryption

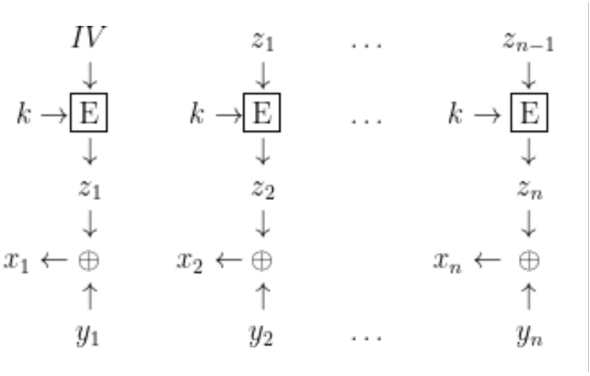


Figure 50: OFB: Decryption

when performing the XOR: ciphertexts are XOR-ed with the key stream to recover the plaintexts.
Notice that the generation of the key stream is, in fact, CBC encryption of a zero plaintext (indeed, notice that in this case we do not have the top level x_i XOR y_{i-1}). It is thus possible to reuse CBC implementations to compute it. Notice also that the plaintext blocks can be smaller that the size of the block cipher. In that case it is possible to use part of the key and use the remaining part for the next block. For example, if the size of the block is 128 bits (like in AES), and we have to encrypt single bytes we have that one key can be split into $128/8 = 16$ keys of 8 bits, each used to encrypt a single byte.

Advantages

- This cipher is very efficient (key can be precomputed using CBC) and allows for the encryption of streams of plaintexts;
- Key stream is generated through a block cipher which makes it very hard to be predicted;
- Finally, notice that it works very well when IV changes every time we encrypt a new block.

Disdvantages

- This stream cipher is synchronous since the key stream is independent of the plaintext. As a consequence, if we reuse the same IV with the same key we obtain the

same key stream. Since encryption is XOR, attacking the cipher is the same as attacking one-time-pad when the key is used more than once. Thus, the IV must be changed any time we encrypt a new message under the same key k;

- Moreover, an attacker in the middle can arbitrarily manipulate bits of the plaintext by swapping the corresponding bits in the ciphertext. No decrypted blocks will be corrupted. For this reason this mode should only be used in application where integrity of the exchanged message is not an issue or is achieved via additional mechanisms. An example could be satellite transmissions where an attacker is extremely unlikely to be in the middle and confidentiality is the only issue. In this setting, absence of integrity becomes useful to avoid noise propagation: an error on one bit will only affect one bit of the plaintext.

NOTE: there exists a variation of OFB, called **Counter mode** (**CTR**), where IV is a random number (nounce) and a counter. The random number can be sent in clear (the bit should change and any new stream generation), and the counter changes the value during the stream generation. This mode is widely used in practice.

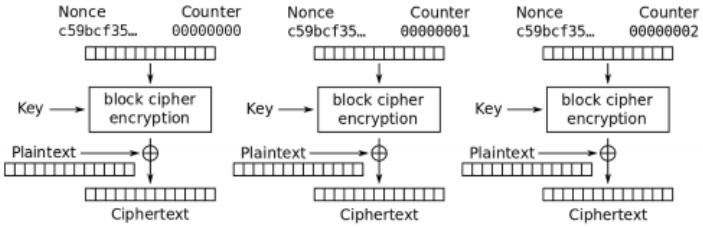


Figure 51: CTR: Encryption

3.3.4 Cipher FeedBack mode (CFB)

This mode mitigates the problems of OFB by making the key stream dependent on the previous encrypted element. To preserve the ability of encrypting plaintexts of size less than or equal to the size of the block of the cipher (e.g. a single byte), this mode uses a shift register that is updated at each step: the register is shifted to the left the number of bits of previous ciphertext (8 for a byte), and such a ciphertext is copied into the rightmost bits of the register.
In this sense, in the first phase we use IV, and in the following ones we use blocks of the previous outputs (IV$|y_1$ or $y_{n-2}|y_{n-1}$). Decryption is as follows:
As for OFB, the key stream is generated and then XOR-ed to the ciphertexts to reconstruct the plaintexts.
Advantage: this mode provides a higher degree of integrity with respect to OFB: whenever one bit of one ciphertext is modified, the next BSize/CSize plaintexts are corrupted, where BSize is the size of the block of the cipher (e.g., 128 bites) and CSize is the size of the single ciphertext (e.g., 8 bits). For example, with AES and 8 bits of plaintext/ciphertext sizes, we have $128/8 = 16$ corrupted decryptions. This number corresponds to the number of left shifts necessary for a ciphertext to exit the shift register.
Disadvantage: on the other hand, this cipher is slower than OFB as it requires the previous ciphertext to compute the next, meaning that parallelization is impossible when

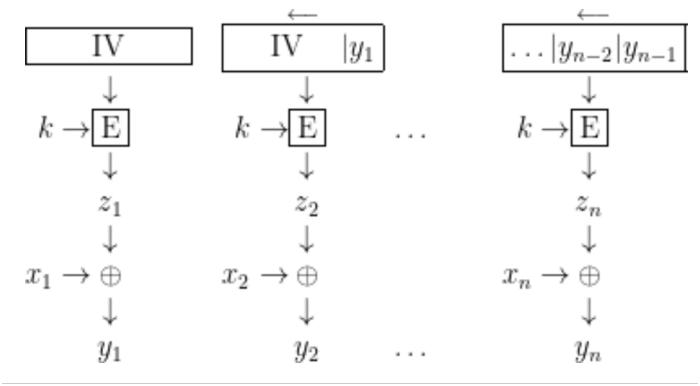


Figure 52: CFB: Encryption

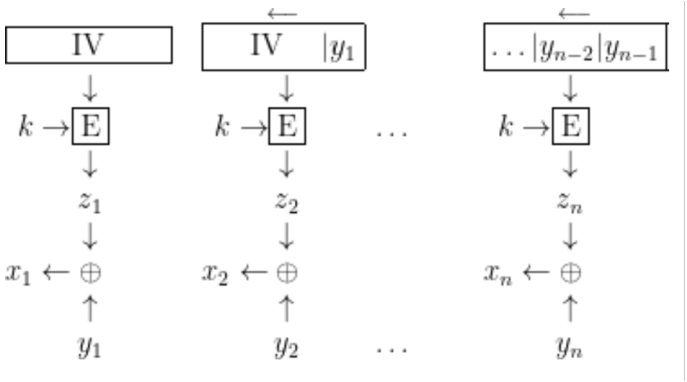


Figure 53: CFB: Decryption

encrypting. Moreover, for noisy transmissions (e.g., satellite, TV, ..) it has the problem of propagating an error on a single bit over the next BSize/CSize plaintexts, which are completely corrupted.

3.4 More block ciphers

There are many other ciphers in use, in addition to AES. We list some here giving a very brief summary of their features.

3.4.1 Data Encryption Standard (DES)

This is the predecessor of AES. It has been published in 1975 and derives from Lucifer (IBM). It has been the most used and implemented cipher in the history and it is currently used in many application, especially in the triple version below (this version uses 3 keys of 56 bits, which make it invulnerable to brute force attacks).
DES major problem is the key-length (only 56 bits) that is considered vulnerable with modern parallel computers. Indeed, notice that this key length ony generates 2^{56} possible keys, so it is prone to brute force attacks. There are also some analytical results which demonstrate theoretical weaknesses in the cipher, although they are infeasible to mount in practice.

Operations The DES cipher takes a fixed-length string of plaintext bits and transforms it through a series of complicated operations into another ciphertext bit string of the same length. We have that:

- The block size is 64 bits;
- Key is of 56 bits (8 for error correction)

The operations are the following:

- 16 identical rounds;
- Initial permutation (IP) and final permutation (FP, inverse operation);
- Before the main rounds, the block is divided into two 32-bit halves and processed alternately:
  - The first half is XOR-ed with the result of F, and the result is given as input to the F of the following round;
  - The second half is provided as input to the F function, and it is also XOR-ed with the result of the F function of the following round.

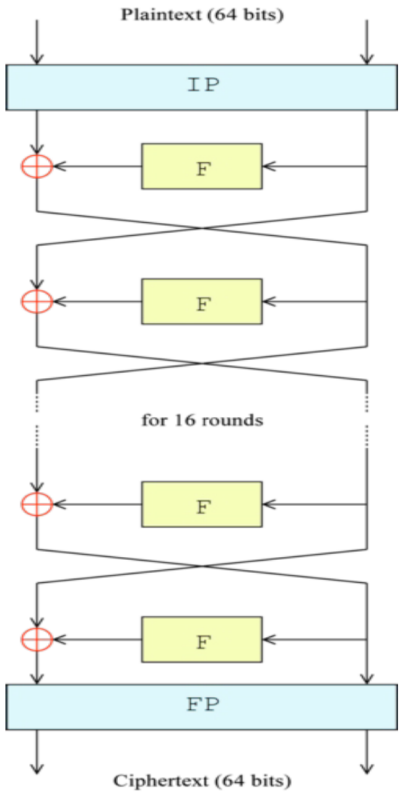


Figure 54: DES: operations

Feistel function This function is composed of the following operations:

1. E: permutation and expansion (from 32 to 48 bits);
2. Key-mixing (key schedule), where the 48 bits are XOR-ed with a subkey of 48 bits, extracted from the original key (56 bits) using permutation and circular shifts;
3. Substitution using S-box, and compression (results 32 bits). In this case, from 8 blocks of 6 bits we retrieve 8 blocks of 4 bits;
4. P: permutation.

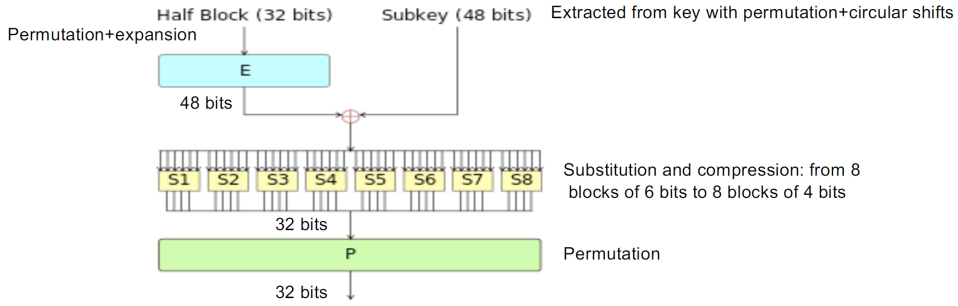


Figure 55: F function

The result of the F function is a 32-bit information, which is consistent with the previous schema, since it is XOR-ed with half of the original block of 64 bits. The alternation of substitution from the S-boxes, and permutation of bits from the P-box and E-expansion provides so-called "confusion and diffusion" (Shannon).

Confusion and diffusion

- Confusion: the process drastically changes data from the input to the output (e.g., by translating data through a non-linear table created from the key);
- Diffusion: changing a single character of the input will change many characters of the output.

3.4.2 International Data Encryption Algorithm (IDEA)

This cipher was proposed in 1990 as a substitute of DES. It is currently adopted in many applications.
This cipher is not based on non-linear substitutions (S-Boxes); instead, confusion and diffusion are obtained by a combination of three operations: XOR, sum and multiplication modulo 216. Patent issues have reduced the popularity of this cipher. Compared to other, IDEA performance is not so high.

3.4.3 Blowfish and Twofish

Blowfish has been proposed in 1993. It is a cipher with peculiar features: it is very fast, compact and simple to implement, with a very highly configurable security: key length is variable up to 448 bits which allows for security/speed trade-off. As DES it is based on XOR and S-Boxes which are not fixed but computed using the cipher itself and the actual key. These key-dependent S-Boxes make brute-forcing particularly expensive: for each key it is necessary to generate the S-Boxes which takes 522 iterations of the algorithm. Twofish is one of the finalists of AES and is the "successor" of Blowfish. Both ciphers have been developed by Bruce Schneier.

3.4.4 RC2, RC5, RC6

This is a family of ciphers developed by Ron Rivest (one of the fathers for RSA public-key cipher). RC5 (1994) has the peculiar feature of using data dependent rotations. Moreover, the cipher is extremely simple but requires a complex key-expansion procedure: each round is just two XORs, two sums modulo and two rotations. This cipher is highly configurable on the number of rounds, key-length and word-length, which allows for a sophisticate trade-off between security and performance. RC6 has been one of the AES finalists.

3.5 Meet-in-the-middle attack - 3DES

One technique to strengthen ciphers is iteration. We have seen that all modern ciphers are based on rounds, i.e., repetitions of the same core algorithm. We might wonder what happens if we iterate a whole cipher such as DES or AES. There is at least a good reason for that: increasing key length. DES, for example, has a 56-bit key that is considered weak nowadays. If we iterate the cipher using a different key we obtain a key pair (k1,k2) of 112 bits which is, in principle, too hard to break (but we will see that this is not the case).

3.5.1 3DES

It is a triple iteration of DES. The aim is to increase the key-length. Due to the meet-in-the-middle attack, the triple key of 168 bits is, in fact, equivalent in strength to a key of 112 bits. Meet-in-the-middle is also the reason why 2DES makes no sense: the 112-bit key could be broken in a 256 time/space complexity brute force attack. 3DES is implemented, for example, in SSH, TLS/SSL and is adopted in many commercial applications. Moreover, bank circuits and credit card issuers use it in smartcard based applications and for PIN protection.

DES is non-idempotent We know that iteration makes sense only if the cipher is not idempotent, otherwise the result of its composition would be the same cipher, so it would be useless. The following informal argument suggests that modern ciphers are very unlikely to be idempotent. We reason on DES but the same reasoning would apply to different block ciphers.

DES has a block size of 64 bits. If we list all the 2^{64} possible blocks and we pick one DES key, the cipher will map each of these blocks into a different block. Since encryption must be invertible, this mapping is injective. Thus, in any block cipher, a key corresponds to a permutation of all the possible plaintext blocks.

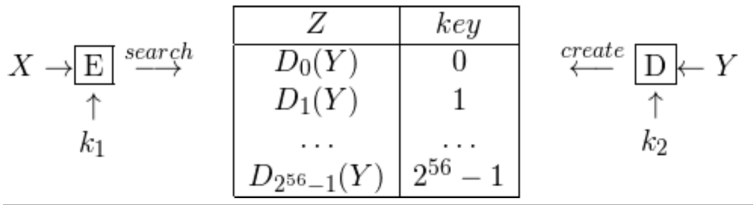
$$\begin{array}{cccccc} & 0 & 1 & 2 & \dots & 2^{64}-1 \\ k & \downarrow & \downarrow & \downarrow & \dots & \downarrow \\ & \rho(0) & \rho(1) & \rho(2) & \dots & \rho(2^{64}-1) \end{array}$$

So, for example, 0 (64 zero bits) is mapped to 3214112, 1 to 213210312421 and so on.
Now, the number of permutations of 2^{64} elements is $2^{64}!$ which is enormously big compared to the 2^{56} DES keys. We reason as follows: the way a DES key selects a specific permutation is "complex" (otherwise the cipher would be weak). We can thus think of DES keys as selecting a random subset of 2^{56} permutations among the $2^{64}!$ possible ones. Now, the probability that the composition of two such permutations is still in this subset is, intuitively, $2^{56}/2^{64}!$ which is a very small, negligible number. This means that it is really unlikely that 2 iterations of DES (and of any modern block cipher, in fact) correspond to a single encryption under a different key.
As far as DES is concerned, it has been formally proved that it is not idempotent.

Meet-in-the-middle We thus consider DoubleDES (2DES), i.e., the iteration of DES twice.
Is it really true that now, in order to break the cipher, we have to try all the possible key pairs?
The answer is 'NO'. We can do better by exploiting the so called Meet-in-the-middle attack. It is a known-plaintext scenario, i.e., the attacker knows pair of plaintext/ciphertext $(X, Y), (X', Y'), (X'', Y''), ..$ all encrypted under the same key K. The idea is:

1. Select one pair, say (X, Y), and try to decrypt Y with all the possible second keys $k2$;
2. All the resulting values Z are stored into a table together with the key, which is indexed by Z;
3. Now we try to encrypt under all the possible first keys $k1$ the plaintext X and we look the obtained value into the table. If we find a match we test the resulting pair $(k1, k2)$ on all the other plaintext/ciphertext pairs and, if all the tests succeeds, we give it as output.

The attack is illustrated below:

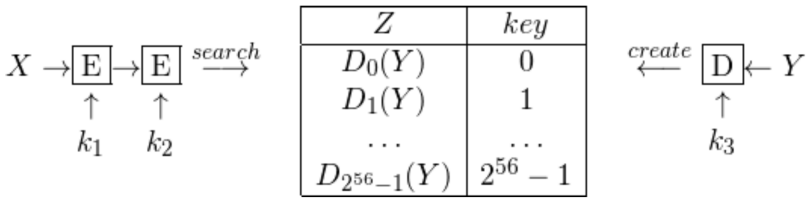


The computational cost of this attack is 2^{57} steps and 2^{56} space. In fact, first step takes 2^{56} steps to build a table which is 2^{56} entries. Second step takes at most 2^{56} steps to find the right key. We thus have $2^{56} + 2^{56} = 2^{57}$ steps.

False keys It is very important that, whenever a pair $(k1, k2)$ for (X, Y) is found, it is tested against other pairs (X', Y'). It could be the case, in fact, that a key pair is fine for (X, Y) but it is not the right key pair. This can happen more frequently than expected.
To estimate the number of these false keys we assume that plaintexts are mapped to ciphertext uniformly by the possible keys. In other words, the number of keys mapping X into Y is approximatively the same as the number of keys mapping X into any other ciphertext Y'. This assumption typically holds for any good cipher for which observing Y gives very little information about the plaintext X. Having a non-uniform distribution would imply that the plaintexts mapped by more keys into Y are more likely than the ones mapped by less keys.
Under this assumption, we can then estimate the number of false keys as $|K|/|C|$, i.e., the number of keys divided by the number of ciphertexts which is, for 2DES, $2^{112}/2^{64} = 2^{48}$. This huge number of possible keys encrypting X into Y can be reduced very quickly by testing keys on more pairs. The probability that a false key is also OK for (X', Y') is just 1 over the number of all the possible ciphertexts (we have only one good case Y' over all the possible 2^{64} ciphertexts) giving $1/2^{64}$ (which is the result of $2^{48} * 1/2^{64}$). Thus, the number of false keys is reduced to $2^{48}/2^{64} = 1/2^{16}$. If we try on one more pair we get $1/2^{80}$, and so on. In summary, with 3 available pairs of plaintext/ciphertext we can run the attack having a negligible probability of getting a false key.

3.5.2 Recap

The cost in time is thus basically the same as the one for a single iteration of DES. For this reason, 2DES is never used in practice and, instead, we have a triple iteration known as triple-DES (3DES). This gives a 168-bit triple key $(k1, k2, k3)$. The meet-in-the-middle attack is still possible but it reduces the cost in time to 2112 with a table of size 256 entries. The idea is to build the table by decrypting Y under all $k3$ and then try all the pairs $(k1, k2)$, as illustrated below.



3.6 Asymmetric-key ciphers

All the ciphers we have studied so far use the **same key** K both for **encryption** and **decryption**. This implies that the source and the destination of the encrypted data have to **share** K. For this reason, this kind of ciphers are also known as **symmetric-key ciphers**. This aspect becomes **problematic** if we want cryptography to **scale** to big

systems with many users willing to communicate securely. Unless we have a centralized service to handle keys (that we will discuss later), for N users this would require $N(N-1)/2$, i.e., $O(N^2)$, keys. For example, for a LAN with 1000 users we would have ≈ 500000 keys. These keys should be pre-distributed to users in a secure way (e.g., offline). This is totally **impractical** and would never scale on a wide-area network such as the Internet. The above argument has been one of the main motivation leading to the **development** of **asymmetric-key cryptography**. In **1976** Diffie and Hellman suggested the existence of this revolutionary ciphers, which are based on the idea that a user A has one **encrypting** and one **decrypting key**, which are **different** but **correlated** (this is why they are called asymmetric). The characteristic is that the **encrypting key** is **public**, while the **decrypting key** is a **secret** known only by A. In this sense, for N users we have N **public** keys, which are used for **encryption**, and N **private** keys, which are used for **decryption**. The public key is published in a public list and is known by everybody, even the attacker, and they are correlated with private keys, but the knowledge of the public key does not give any information about the private key.

3.6.1 Definition

Intuitively, if we denote with PK_A the public key of A and with SK_A the secret key of A, then:

- B sends an encrypted message $E_{PK_A}(M)$ to A;
- A receives it and decrypts it as $D_{SK_A}(E_{PK_A}(M)) = M$.

Moreover, encryption and decryption algorithms are such that

$$D_{SK_A}(E_{PK_A}(M)) = M$$

holds. We can notice that any user can perform the encryption E_{PK_A}, so our goal is to define a decryption function that is unfeasible to broke, otherwise the cipher would be useless.
More formally, an asymmetric-key cipher is a **quintuple** $(\mathcal{P}, \mathcal{C}, \mathcal{K}_S \times \mathcal{K}_P, E, D)$ with $E : \mathcal{K}_P \times \mathcal{P} \to \mathcal{C}$ and $D : \mathcal{K}_S \times \mathcal{C} \to \mathcal{P}$ (*which was the tuple for symmetric ciphers?*) and such that:

1. It is **computationally easy** to generate a **key-pair** $(SK, PK) \in \mathcal{K}_S \times \mathcal{K}_P$;
2. It is **computationally easy** to compute $y = E_{PK}(x)$;
3. It is **computationally easy** to compute $x = D_{SK}(y)$;
4. **Decryption** under SK of a plaintext x **encrypted** under PK gives the initial **plaintext** x. Formally, $D_{SK}(E_{PK}(x)) = x$;
5. It is **computationally infeasible** to compute SK knowing PK and y;
6. It is **computationally infeasible** to compute $D_{SK}(y)$ knowing PK and y and without knowing SK;

Intuitively, instead of one key we now have a **key pair** (SK, PK) composed of a **private** and a **public** key, respectively. **Encryption** is performed under PK while **decryption** under SK. Thus, **decryption key** is now **different** from **encryption key**.
Items 1,2 and 3 state that it should be **practical** to **generate key pairs** and to encrypt/decrypt data.
Item 4 states that **decrypting** under the **private key** a **plaintext encrypted** under the **public key** gives the initial **plaintext**.
Items 5 and 6 state that the **cipher** should be **secure** regardless of the secrecy of the public key PK. This is the **main challenge** behind this kind of cryptography: the **public** and the **private** key have to be **related**, otherwise decryption would never work, but, at the same time, **computing** the **private key** from the public key should be **impossible** in practice (computationally infeasible).

3.6.2 Security properties

What security properties do we have?

- **Secrecy**: this property clearly holds, since we're taking into consideration a cipher;
- **Authentication**: this property does not hold, since the receiver cannot retrieve the identity of the sender, because the key for encryption is public, i.e. known to everybody. Thus, in this case to ensure this property we would also need a digital signature.

Notice that in the case of symmetric cipher, both the properties hold, since in that case each sender/receiver share a single key, so the authenticity is ensured.

3.6.3 One-way trap-door functions

Asymmetric-key ciphers are strictly **related** to **one-way trap-door functions**.
Definition. An **injective**, **invertible function** f is **one-way**, if and only if:

1. $y = f(x)$ is **easy** to compute (i.e. encryption is computationally easy);
2. $x = f^{-1}(y)$ is **infeasible** to compute (i.e. decryption is computationally hard).

Note that one-way does not refer to the fact the function does not admit an inverse.: **one-way** functions are **invertible** but **computing their inverse** is too **expensive** to be feasible in practice.
Definition. An **injective**, **invertible family of functions** f_K is **one-way trap-door**, if and only if, given K, we have that:

1. $y = f_K(x)$ is **easy** to compute;
2. $x = f_K^{-1}(y)$ is **infeasible** to compute **without knowing the secret trap-door** $S(K)$ relative to K.

Thus, intuitively, the trap-door is a hidden way to go back to the pre-image x of the function: **only knowing the trap-door we can compute the inverse** of f_K.

3.6.4 The Merkle-Hellman knapsack system

We present now an example cipher that has been broken, but still gives a very immediate idea of how asymmetric-key ciphers relate to one-way trap-door functions. The cipher is based on the following NP-complete problem.

The subset-sum problem Let $s_1, .., s_n$ and T be positive integers: s_i are **sizes** while T is the **target**. A **solution** to the subset-sum problem is a **subset** of $(s_1, .., s_n)$ whose **sum** is exactly the **target** T.
Formally, the solution is a binary tuple $(x_1, .., x_n)$ such that $\sum_{i=1}^{n} x_i s_i = T$.
For example, if sizes are $(4, 6, 3, 8, 1)$ and $T = 11$, we have that $(0, 0, 1, 1, 0)$ and $(1, 1, 0, 0, 1)$ are solutions, since $3 + 8 = 11$ and $4 + 6 + 1 = 11$.
This **problem** is **NP-complete** in general. As a consequence, we can easily obtain a one-way function from it. Notice, in fact, that NP problems have a non-deterministic polynomial solution meaning that checking if a solution is correct can be done in polynomial time. If we define $f(x_1, .., x_n) = \sum_{i=1}^{n} x_i s_i$, we have that f is clearly easy to compute but **inverting** this function amounts to **finding** $(x_1, .., x_n)$ from a **target** T which we know to be **infeasible** for a big n.
How can we now introduce a secret trap-door to allow us inverting the function? The trick is to start from a specific instance of the problem that is easy to solve. We consider special **sizes** that are **super-increasing**, i.e., such that $s_i > \sum_{j=1}^{i-1} s_j$, for each $i > 1$. Intuitively, any s_i is bigger than the sum of all the previous s_j. For example (1,3,5,10) is super-increasing while (1,3,5,9) is not. In this special case there is a very **efficient algorithm** to solve the **subset-sum problem**. The idea is to **start** from the **biggest element** s_n and go back to the first one: if s_i fits into T we pick it (we set $x_i = 1$) and we subtract s_i from T. Python example code follows.

```
def subsetSum(S,T):  # assumes S is a super-increasing list of integers
    x = []
    S.reverse()          # reverse list to start from the biggest
    for s in S:          # iterates on all s_i (from the biggest)
        if s <= T:
            x.append(1)  # takes the element
            T = T-s      # subtracts it from T
        else:
            x.append(0)  # does not take the element
    if T==0:             # solution found
        x.reverse()
        return x         # returns the reversed tuple
    else:
        return []        # no solution found
```

Figure 56: Solution of the subset-sum problem with super-increasing sizes

Example: subsetSum($[1, 3, 5, 10]$, 11) returns $[1, 0, 0, 1]$, while subsetSum($[1, 3, 5, 10]$, 12) returns the empty list [], since there's no solution in this case.

The cipher We now proceed as follows:

1. We start from a **super-increasing** problem $(s_1, \ldots, s_n)$;

2. We choose a **prime** $p > \sum_{i=1}^{n} s_i$;
3. We choose a **random** a such that $1 < a < p$;
4. We **transform** the initial super-increasing problem into $(\hat{s_1}, \ldots, \hat{s_n})$, with $\hat{s_i} = as_i \mod p$. Notice that **this problem** is **not super-increasing in general**;

The **trap-door** is composed of the **initial problem** and values p and a, that are kept **secret**. Intuitively, **encryption** is done by **computing the target** for the **problem** $(\hat{s_1}, \ldots, \hat{s_n})$. Since this is not super-increasing, finding the initial plaintext $x_1, \ldots, x_n$ is **infeasible** (for a big n). However, **knowing** the **secret trap-door** we can **derive** from the **ciphertext** a **target** for the initial easy super-increasing problem, and then solve it using the efficient algorithm above.
More precisely, **encryption** and **decryption** are defined as follows:

- $E_{PK}(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i \hat{s_i}$;
- $D_{SK}(y)$ is the solution of the super-increasing problem $(s_1, \ldots, s_n)$ with target $a^{-1}y \mod p$.

Notice that $a^{-1} \mod p$ is guaranteed to **exist** by the fact p is a **prime** number. We will prove this in the next lesson.
The **correctness** of the above cipher can be proved as follows:

$$\begin{array}{rcl} E_{PK}(x_1, \ldots, x_n) & = & \sum_{i=1}^{n} x_i \hat{s_i} \\ & = & \sum_{i=1}^{n} a x_i s_i \mod p \\ & = & a \sum_{i=1}^{n} x_i s_i \mod p \end{array}$$

, thus

$$\begin{array}{rcl} a^{-1}E_{PK}(x_1, \ldots, x_n) \mod p & = & a^{-1} a \sum_{i=1}^{n} x_i s_i \mod p \\ & = & \sum_{i=1}^{n} x_i s_i \mod p \\ & = & \sum_{i=1}^{n} x_i s_i \end{array}$$

, since we have that $p > \sum_{i=1}^{n} s_i$ by construction.
The previous equations show that the **transformed target** $a^{-1}y \mod p$ is, in fact, the **target** for the **initial**, easy **problem**.
Example: let $(1, 2, 5, 12)$ be a super-increasing problem. We let $p = 23$ and $a = 6$. We have that $a^{-1} = 4$, since $6 \cdot 4 \mod 23 = 1$. If we compute $s_i \cdot a \mod p$ we obtain $(6, 12, 7, 3)$ which is not super-increasing. Consider now the plaintext $(1, 0, 0, 1)$. We have that $E_{PK}(1, 0, 0, 1) = 6 + 3 = 9$. Now to decrypt we have to compute $a^{-1} \cdot 9 \mod p = 36 \mod 23 = 13$. We finally solve $(1, 2, 5, 12)$ with target 13, which gives the initial plaintext $(1, 0, 0, 1)$.

3.7 The RSA cipher

RSA is the most famous **asymmetric-key cipher**. It is based on a few technical results from number theory that we recall below.

3.7.1 Background

The Euler function The Euler function $\Phi(n)$ returns the number of numbers less than or equal to n that are coprime to n. Recall that i and n are coprime iff $gcd(i, n) = 1$, i.e., if the only common divisor is 1. So, for example, we have that $\Phi(3) = 2$ since 1 and 2 are coprime to 3, $\Phi(4) = 2$ since only 1 and 3 are coprime to 4 and so on. We report some values below:

n	$\Phi(n)$
1	1
2	1
3	2
4	2
5	4
6	2
7	6

We immediately notice that if n is prime then $\Phi(n) = n - 1$. In fact, by definition, a prime number n is coprime to all the numbers smaller than n.
There is another situation where $\Phi(n)$ is easy to compute: when $n = p_1 \dots p_k$ with $p_1 \neq p_2 \neq \dots \neq p_k$ prime numbers, i.e. multiplication between prime numbers, we have that $\Phi(n) = \Phi(p_1) \dots \Phi(p_k) = (p_1 - 1) \dots (p_k - 1)$.
Example: consider $15 = 3 \cdot 5$, then $\phi(15) = \phi(3) \cdot \phi(5) = 2 \cdot 4 = 8$.
We prove this result for $n = pq$ with $p \neq q$ primes (i.e. we want to prove that $\phi(n) = \phi(p)\phi(q) = (p-1)(q-1)$). The numbers less than n that are not coprime to n is exactly the multiples of p and q, i.e., $p, 2p, \dots, (q-1)p, q, 2q, \dots, (p-1)q$ that are $(q-1)+(p-1)$. Now we have that $\Phi(n) = pq - 1 - (q-1) - (p-1) = pq - q - p + 1 = (p-1)(q-1)$.
Example: consider $14 = 2 \cdot 7$. Then, $\phi(14) = 13 - 6 - 1 = 6$, where 6 is the number of multiples of 2 up to 14, while 1 is the only multiple of 7 up to 14.

3.7.2 The cipher

RSA stands for Rivest-Shamir-Adleman, the authors of the cipher in 1978.
We let $n = pq$ with p, q big **prime** numbers (we need a method for generating large prime numbers, we will consider an algorithm): notice that $\phi(n) = (p-1)(q-1)$. We then choose a small a, prime with $\phi(n)$ and smaller than $\phi(n)$.
Example: let $n = 3 \cdot 5$, then $\phi(n) = 2 \cdot 4 = 8$. We can choose $a = 7$.
We compute the unique b s.t. $ab \mod \phi(n) = 1$.
Example: considering the previous example, we have that $b = 7$, since $7 \cdot 7 \mod 8 = 1$ (not a good example in this case, since $a = b$).
The **public key** is (b, n) while the **private key** (secret trapdoor) is (a, n): notice that a is secret, while b and n are known. We let $\mathcal{P} = \mathcal{C} = \mathbb{Z}_n$.
Encryption is defined as

$$E_{PK}(x) = x^b \mod n$$

while **decryption** is

$$D_{SK}(y) = y^a \mod n$$

Example: let $p = 5$ and $q = 11$, then $n = 5 \cdot 11 = 55$ and $\phi(n) = 4 \cdot 10 = 40$. We choose a coprime to 40 and smaller than 40, in this case $a = 23$. Thus, $SK = (23, 55)$. We compute the unique b s.t. $ab \mod \phi(n) = 1$, in this case $b = 7$ (computed with the extended Euclidean algorithm). Thus, $PK = (7, 55)$. If we consider the encryption of $x = 2$, then

$$E_{PK}(2) = 2^7 \mod 55 = 128 \mod 55 = 18$$

, while the decryption of 18 results to be

$$D_{SK}(18) = 18^{23} \mod 55 = 2$$

3.7.3 Implementation

As we will discuss, RSA requires a big modulus n of at least 1024 bits. With these sizes, implementation becomes an issue. For example a linear complexity $O(n)$, that is typically considered very good, is prohibitive as it would require at 2^{1024} steps. Every operation should in fact be polynomial with respect to the bit-size k.
We first observe that basic operations such as **sum**, **multiplication** and **division** can be performed in $O(k), O(k^2), O(k^2)$, respectively, by using simple standard algorithms (the one we use when we compute operations by hand). Reduction modulo n amounts to compute a division which is, again, $O(k^2)$.

Exponentiation This operation is used both for **encryption** and **decryption**. First notice that we cannot implement exponentiation to the power of b as b multiplications. In fact, public and private exponents can be the same size as n. Performing b multiplications would then require $k^2 2^k$ operation, i.e., $O(2^k)$ which is like brute-forcing the secret trapdoor and infeasible for $k \geq 1024$. We thus need to find some smarter, more efficient way to compute this operation.

3.8 Exercises

1. Show that the composition of the shift cipher with the substitution cipher is still a substitution cipher with a different key. Give a constructive way to derive the new key. What happens if substitution is applied before shift? Solution on slide 23-24-25 of L7;
2. Consider the composition of Vigenére cipher with key ALICE with the shift cipher with key 8. Is the resulting cipher equivalent to a known one? If so, what is the resulting key? Solution on slide 28 of L7;
3. Show that the composition of Vigenére and the shift cipher is idempotent. Solution on slide 34 of L7;
4. Multiply $(x^4 + x^3 + 1) \times (x^5 + 1)$. Solution on slide 17 of L8;
5. Multiply $(x^4 + x^3 + 1) \times (x^5 + 1)$ using the optimized algorithm. Solution on slide 47 of L8;
6. Multiply $(x^4 + x) \times (x^3 + 1)$ using the optimized algorithm. Solution on slide 39 of L9;
7. Multiply $(x^5) \times (x^3 + 1)$ using the optimized algorithm. Solution on slide 41 of L9;
8. Encode the plaintext "Two One Nine Two" with AES, with the key being "Thats my Kung Fu". Solution on slide 21 to 37 of L9;
9. Write the expressions for CBC encryption and decryption of the i-th block and show, formally, that $D_k^{CBC}(E_k^{CBC}(x_i)) = x_i$.

   Hint: to avoid defining a special expression for y_1, you can let $y_0 = IV$.

   Solution on slide 2 of L10;
10. Let $(2, 7, 11, 21, 42, 89, 180, 354)$ be a super-increasing sequence, $p = 881$ and $a = 588$ (secret key).

    - Compute the public key $s_i \cdot a \mod p$;
    - Then compute the encryption and the decryption of letter “a” (look at the ASCII binary encoding of the letter).

    Solution on slide 16-17 of L12;
11. Generate two distinct prime numbers, $p = 11$ and $q = 13$.

    Compute possible $PK, SK, EPK(x), DSK(y)$ with $x = 2$. Solution on slide 31-32 of L12.