
# 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.
**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.

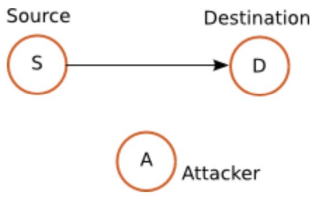



Figure 1: Expected information flow

### 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:

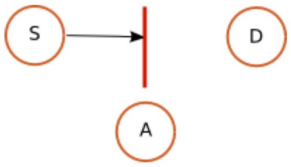



Figure 2: The attacker interrupts the flow of information

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:

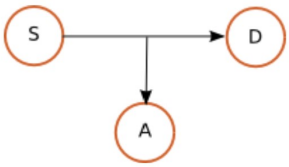



Figure 3: The attacker intercepts information

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:
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.

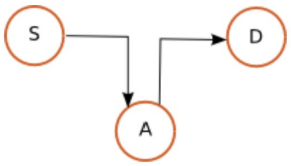



Figure 4: The attacker modifies information

### 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:

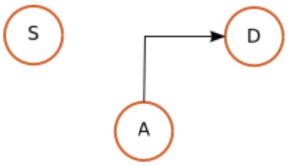



Figure 5: The attacker forges new information

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.

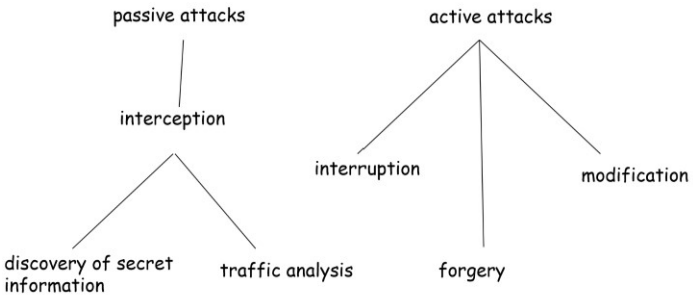



Figure 6: Passive and active attacks

### 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):

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

,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.

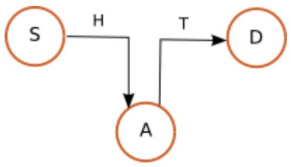



Figure 8: The honest program $H$ is modified into a Trojan $T$

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.

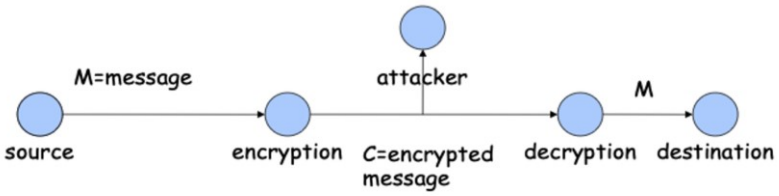



Figure 9: First encryption strategy

   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:

   - 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**.

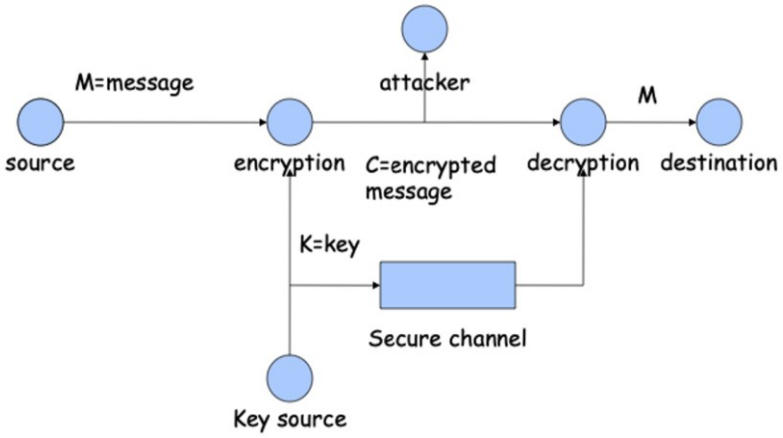



Figure 10: Second encryption strategy

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

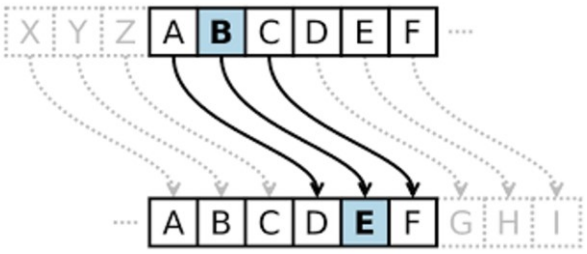



Figure 11: Caesar cipher

alphabet. So, for example, 'A' is replaced with 'D' and 'M' with 'P'. The substitution can be represented as follows:
, 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**.

Encryption:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

DEFGHIJKLMNOPQRSTUVWXYZABC

Decryption:

DEFGHIJKLMNOPQRSTUVWXYZABC

ABCDEFGHIJKLMNOPQRSTUVWXYZ

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.*

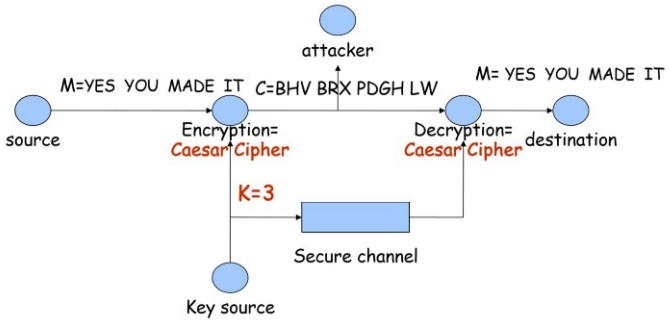



Figure 13: Caesar cipher: example

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 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 $k = 3$.
For example $k = 10$ gives the following substitution (notice that the bottom alphabet is now shifted to the left by 10 positions):

ABCDEFGHIJKLMNOPQRSTUVWXYZ
KLMNOPQRSTUVWXYZABCDEFGHIJ

Figure 14: Shift cipher: example

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

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

An example is provided in Picture 2.2.3.

ABCDEFGHIJKLMNOPQRSTUVWXYZ
SWNAMLXCVJBUYKPDOQERIFHGZT

Figure 15: Substitution cipher: example

Let's see if this cipher satisfy the properties:

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

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

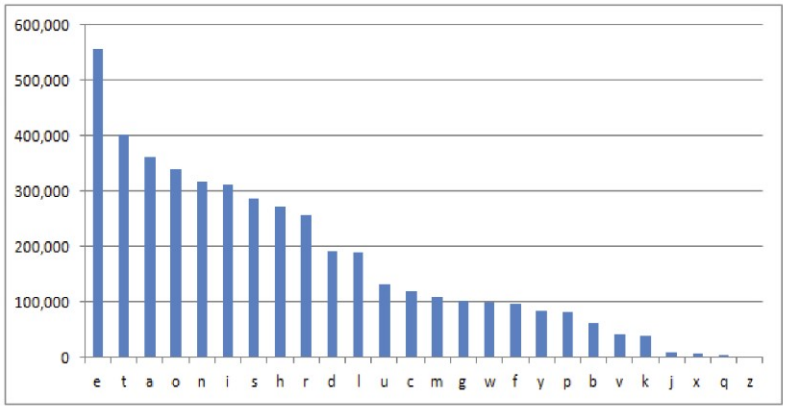
   , so the **first property is satisfied**;
2. Computing $k$ or $x$ given a ciphertext $y$ is unfeasible. Is the brute force technique still possible in this case? Well, we have 26! possible keys (all the possible permutations of 26 elements), which is approximately $4 * 10^{26} > 2^{88}$, which would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

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 | 14998 | 0,998 |




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

Figure 16: Language statistics

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**.

### 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** ($\approx 0.038$) we have a **polyalphabetic cipher**, if it is $\approx 0.065$ 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:
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

```
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 $m$

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.

ULRFCZ DLL VACL GNU GSAEA m=6

FLAUTO FLA UTOF LAU TOFLA

UDLA (all with the shift of F, i.e., 5)

LLGE (all with the shift of L, i.e., 9)

RLNA

FVU

CAG

ZCS

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* $M = message = (x_1, x_2) = (5, 9)$, *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:*

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

*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* $tmp = 12 - 2 * 5 = 2$.

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* $EuclidExt(5, 17) = 7$.

## 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**.

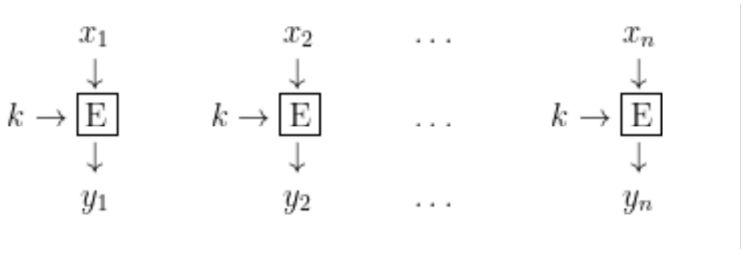



Figure 23: Block cipher

This scheme can be **generalized** by considering a **stream of keys** instead of a fixed one. Let $z_1, z_2, \dots, z_n$ 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 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).

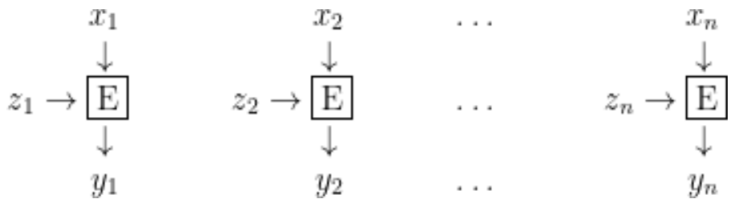



Figure 24: Cipher using different keys

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:

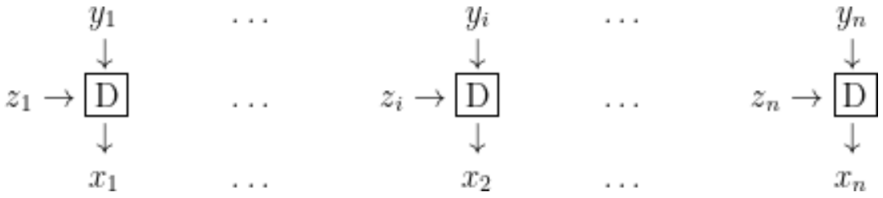



Figure 25: Decryption of a cipher using different keys

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: $z_1, x_1, z_2, x_2, \ldots, z_n, x_n$. 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 $z_1, z_2, \ldots, z_d, z_1, z_2, \ldots, z_d, z_1 \ldots$, 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.

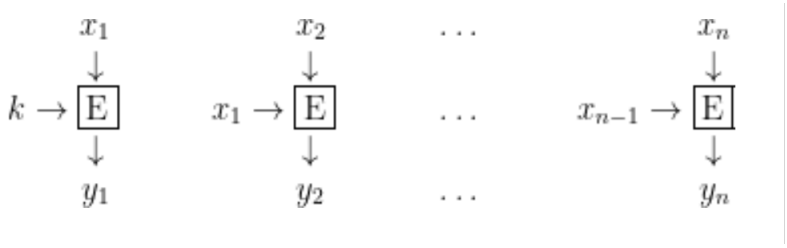



Figure 26: Autokey cipher

**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: We now let* $p_p(a) = 3/4$, $p_p(b) = 1/4$, $p_k(k_1) =$

| $E$ | $a$ | $b$ |
|---|---|---|
| $k_1$ | 1 | 2 |
| $k_2$ | 2 | 3 |

Table 1: Caption

$p_k(k_2) = 1/2$. *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 $1/n$.
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.

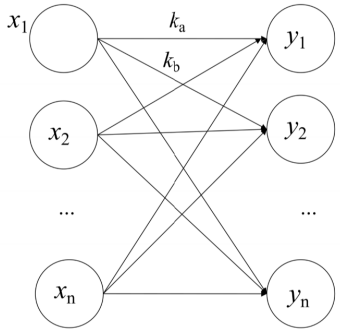



Figure 27: Completely bipartite graph

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.

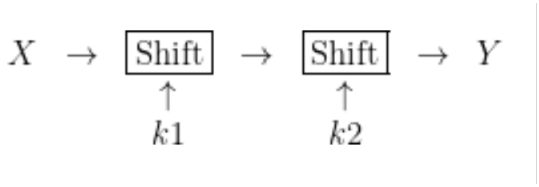



Figure 28: Composition of two shift ciphers - 1

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.

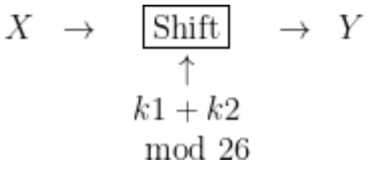



Figure 29: Composition of two shift ciphers - 2

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* $E^1_k(x) =$

$E_k^2(x) = x + k \mod 26$. *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):

| | |
|---|---|
| $(S^1 \times S^2) \times (S^1 \times S^2)$ | |
| $= S^1 \times (S^2 \times S^1) \times S^2$ | associative property |
| $= S^1 \times (S^1 \times S^2) \times S^2$ | commutative property |
| $= (S^1 \times S^1) \times (S^2 \times S^2)$ | associative property |
| $= S^1 \times S^2$ | idempotence of the initial ciphers |

### 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,

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

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.

An example is provided in Picture 3.2.1.

a b p
11 1011 00 XOR 11 = 11
110 101 11 XOR 110 = 101
1100 10
11000 1 101 XOR 11000 =11101 product

$(x+1)x(x^3+x+1)=x^4+x^2+x+x^3+x+1= x^4+x^3+x^2+1$

Figure 31: Product - optimization: example

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.

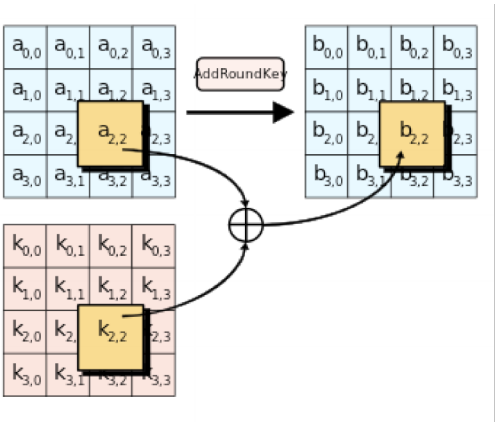



Figure 32: AddRoundKey operation

   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 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:

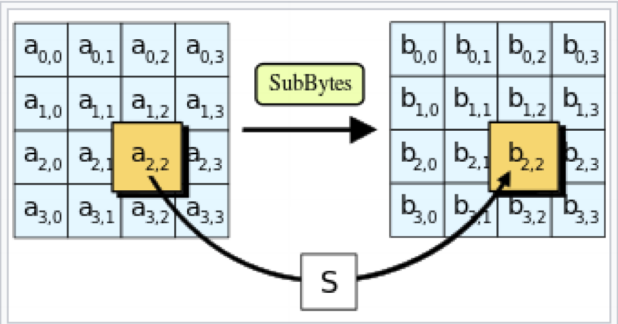



Figure 33: SubBytes operation

| | 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

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

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

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

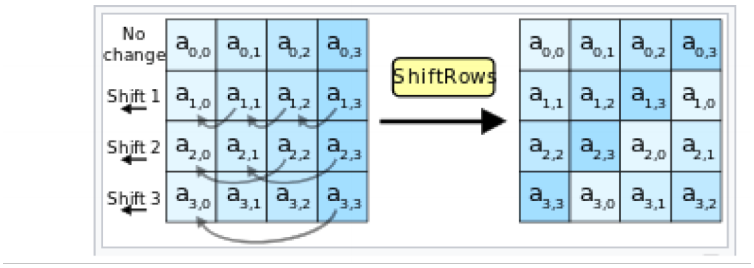



Figure 36: ShiftRows operation

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

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.

   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.

```
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

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.

```
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

Notice that in the **last round we do not perform** the *MixColumn* **operation**.
**Decryption** is computed by **applying inverse operations**.

```
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

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);

| | 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:

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

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

**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 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*(state$\oplus$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.
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.

```
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

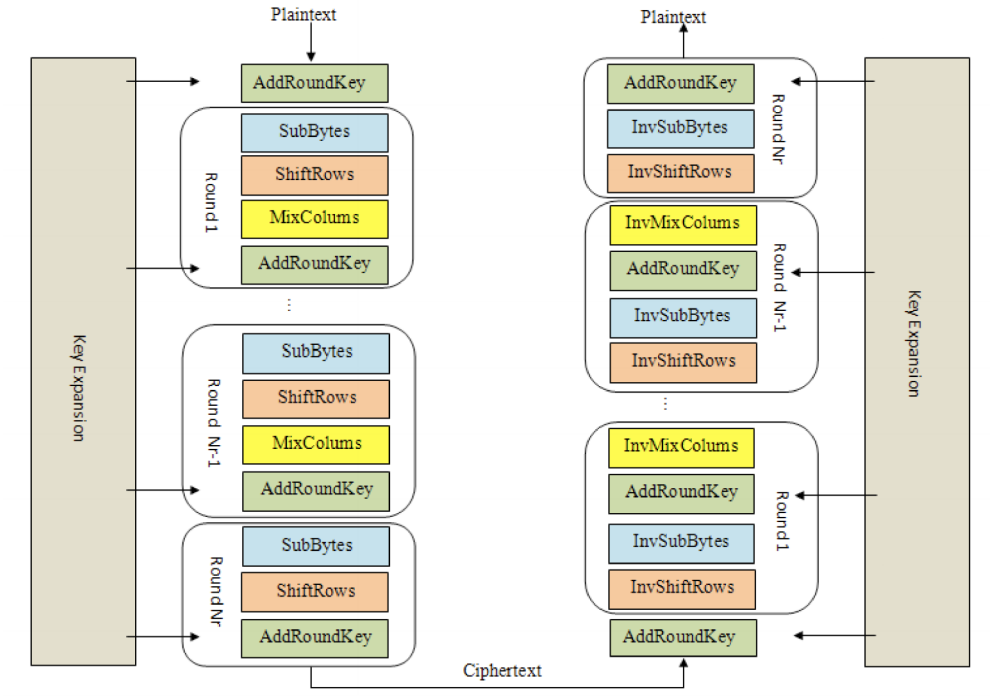



Figure 43: AES algorithm

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 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.

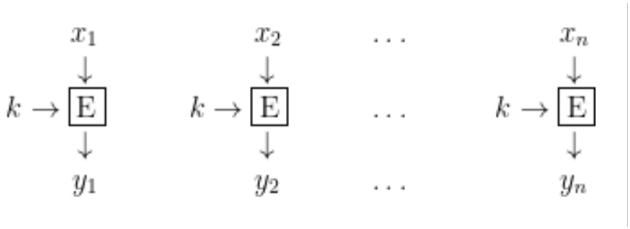



Figure 44: ECB: Encryption

**Decryption** is done, as expected, by reversing the scheme:

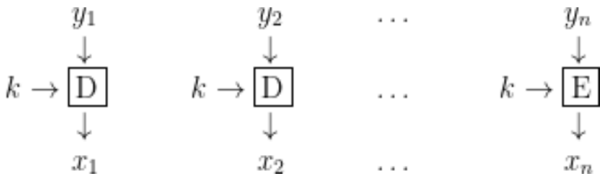



Figure 45: ECB: Decryption

**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:

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.

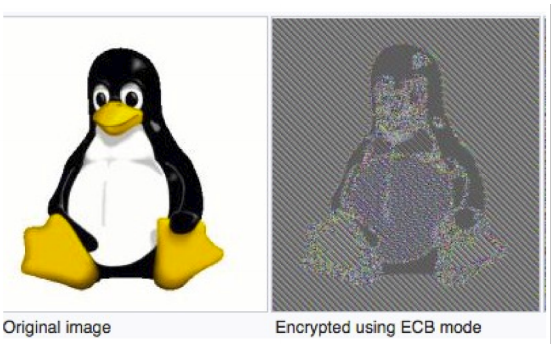



Figure 46: Attack on ECB

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 $k$.

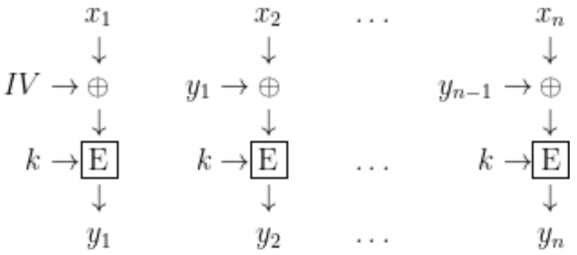



Figure 47: CBC: Encryption

**Decryption** is as follows:

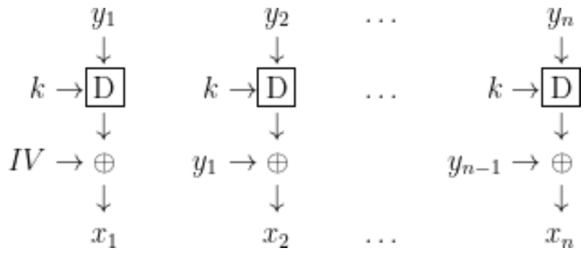



Figure 48: CBC: Decryption

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.

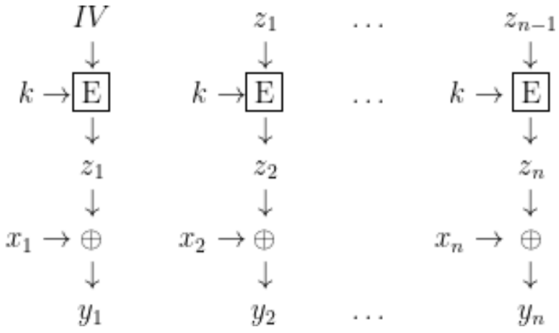



Figure 49: OFB: Encryption

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" when performing the XOR: ciphertexts are XOR-ed with the key stream to recover the plaintexts.

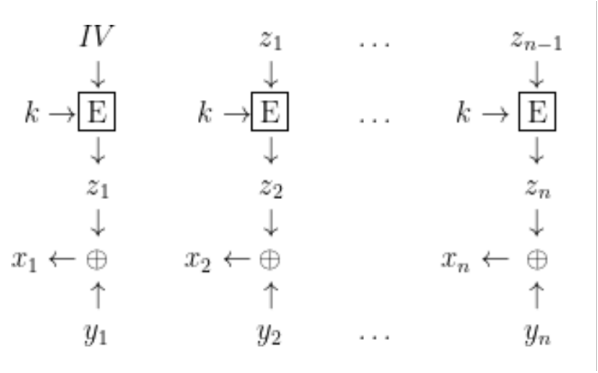



Figure 50: OFB: Decryption

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.

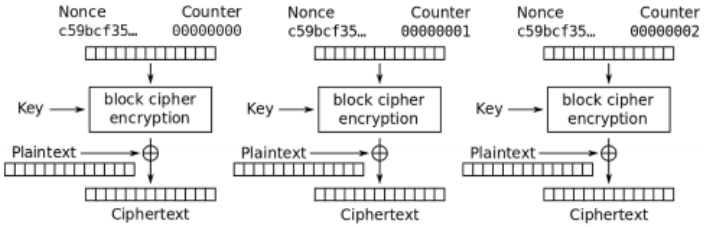



Figure 51: CTR: Encryption

### 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.

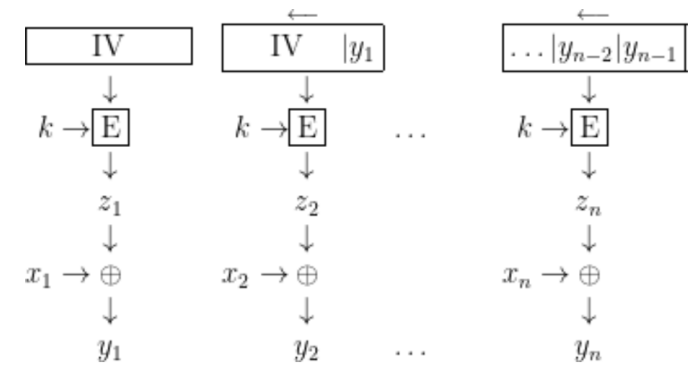



Figure 52: CFB: Encryption

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:

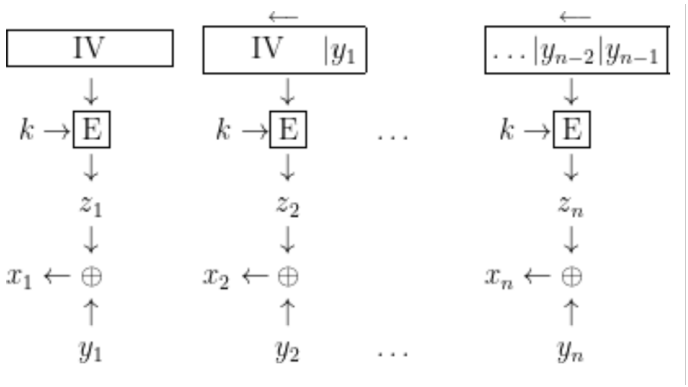



Figure 53: CFB: Decryption

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 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.

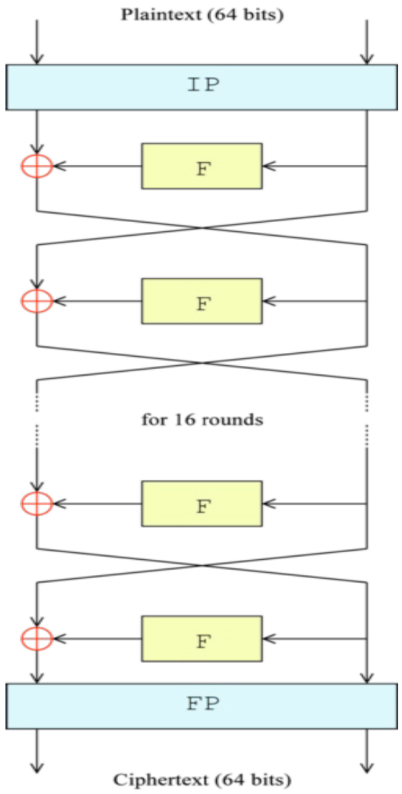



Figure 54: DES: operations

**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;

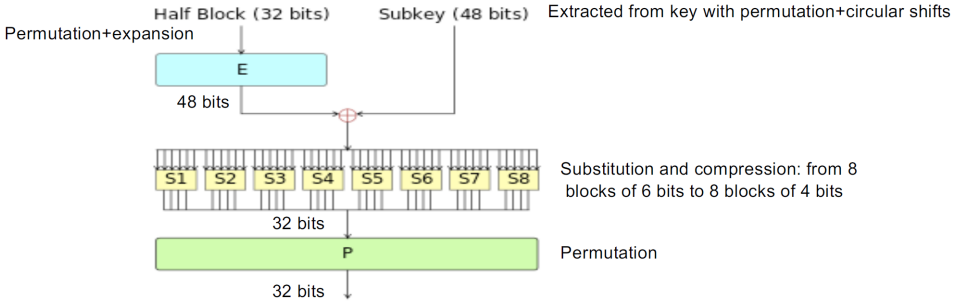



Figure 55: F function

4. P: permutation.

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.

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

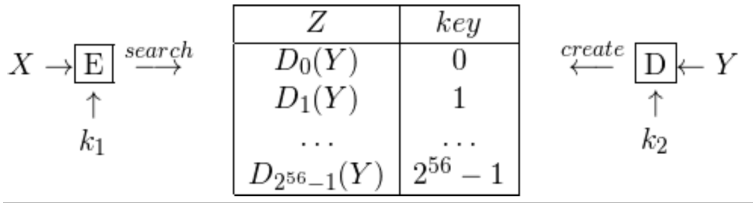
So, for example, 0 (64 zero bits) is mapped to 3214112, 1 to 213210312421 and so on.
Now, the number of permutations of $2^{64}$ elements is $2^{64}!$ which is enormously big compared to the $2^{56}$ DES keys. We reason as follows: the way a DES key selects a specific permutation is "complex" (otherwise 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:

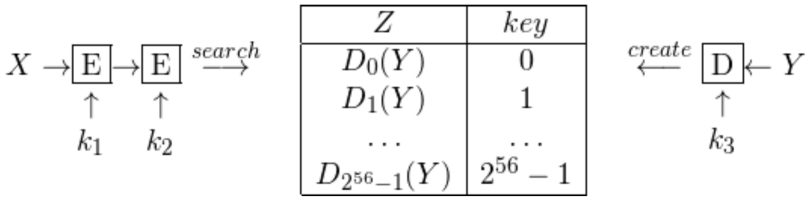



The computational cost of this attack is $2^{57}$ steps and $2^{56}$ space. In fact, first step takes $2^{56}$ steps to build a table which is $2^{56}$ entries. Second step takes at most $2^{56}$ steps to find the right key. We thus have $2^{56} + 2^{56} = 2^{57}$ steps.

**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 $\approx 500000$ 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.*

**Theorem (Euler theorem).** *Let $a$ and $n$ be coprime, i.e., $gcd(a, n) = 1$. Then*

$$a^{\Phi(n)} \mod n = 1$$

*Proof.* Let $S = (s_1, \dots, s_{\Phi(n)})$ be the $\Phi(n)$ numbers less than $n$ and coprime with $n$. We consider $R = (as_1 \mod n, \dots, as_{\Phi(n)} \mod n)$ and we show that $S = R$. We need a few lemmas:

**Lemma 1.** *Let $x, y$ be coprime to $n$. Then $xy$ is coprime to $n$.*

*Proof.* This can be easily proved by considering that all divisors of $xy$ are products of divisors of $x$ or $y$. Thus a common divisor of $xy$ and $n$ must also divide $x$ and/or $y$. Thus, $\gcd(xy, n) > 1$ would imply that $\gcd(x, n) > 1$ or $\gcd(y, n) > 1$ giving a contradiction.

**Example.** *Let $x = 7$, $y = 3$, $p = 5$ and $q = 11$. Then $n = 55$, $x$ and $y$ are coprime to 55, what about $xy = 21$? We have that $gcd(21, 55) = gcd(7 * 3, 5 * 11) = 1$ .*

**Lemma 2.** *Let $x$ be coprime to $n$. Then $x \mod n$ is coprime to $n$.*

*Proof.* Since $x \mod n = x - kn$ we have that any common divisor $d$ of $x \mod n$ and $n$ must divide $x$. In fact, $\frac{x \mod n}{d} = t = \frac{x-kn}{d} = \frac{x}{d} - kr$ that implies $\frac{x}{d}$ is an integer value. That is, $d$ divides $x \mod n$, $n$ and $x$. However, if $x$ is coprime to $n$, then $x \mod n$ is coprime to $n$.

**Example.** *Let $x = 57$, $p = 5$, $q = 11$, then $n = 55$. We have that $x = 57$ is coprime to 55, what about $x \mod n = 2$? We have that $gcd(2, 55) = 1$ .*

**Lemma 3.** *Let $ax \mod n = ay \mod n$ with $gcd(a, n) = 1$. Then $x \mod n = y \mod n$.*

*Proof.* We have $ax - kn = ax \mod n = ay \mod n = ay - jn$. Thus $ax - ay = wn$ that implies $\frac{ax-ay}{a} = \frac{wn}{a}$ and so $x - y = \frac{wn}{a}$ . Since $\gcd(a, n) = 1$ we have that $a$ must divide $w$ and so $x - y = tn$, i.e., $x \mod n = y \mod n$.

**Example.** *Given $a = 3$ and $n = 6$, find $x$ and $y$ such that $ax \mod n = ay \mod n$ (here $gcd(a, n) = gcd(3, 6) = 3 \neq 1$). We have that $x = 1$ and $y = 3$, since $1 \mod 6 = 1 \neq 3 \mod 6 = 3$.*

**Example.** *Find a coprime to $n$, and find $x$ and $y$ s.t. $ax \mod n = ay \mod n$. Let $a = 5$, which is coprime to 6, then $5x \mod 6 = 5y \mod 6$, from which we derive $x = 1$ and $y = 7$. That is $1 \mod 5 = 1 = 7 \mod 6$.*

Now, by Lemma 1 and 2 we have that all numbers in set $R$ are coprime to $n$ and smaller than $n$. Since $S$ is the set of all numbers coprime to $n$ and smaller than $n$ we obtain that $R \subseteq S$. Now, consider $as_i \mod n$ and $as_j \mod n$ in $R$. Since $a$ is coprime to $n$, by Lemma 3 we obtain that $as_i \mod n = as_j \mod n$ implies $s_i \mod n = s_j \mod n$, which gives a contradiction. Thus we have that $as_i \mod n \neq as_j \mod n$ for all $i$ and $j$. This proves that $R = S$.
To conclude the proof it is enough to observe that

$$\prod_{i=1}^{\Phi(n)} s_i = \prod_{i=1}^{\Phi(n)} as_i \mod n = a^{\Phi(n)} \prod_{i=1}^{\Phi(n)} s_i \mod n$$

Since all $s_i$ are coprime to $n$, by applying many times Lemma 3 we obtain $1 = a^{\Phi(n)} \mod n$, which is what we wanted to prove.

**Example.** *Let $p = 2$ and $q = 3$: what happens if $a = 3$ is not coprime to $n = 2 * 3 = 6$? We have that $a^{\phi(n)} \mod n = 3^{\phi(n)} \mod 6 = 3^3 \mod 6 = 3 \neq 1$.*
*We now try to find an $a$ that satisfies the theorem. We let $a = 5$ (5 is coprime to 6), then $5^{\phi(n)} \mod 605^2 \mod 6 = 1$*

### 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 Correctness of RSA

We now show the correctness of the cipher, i.e., that decrypting under the private key a plaintext $x$ encrypted under the public key gives $x$. Notice that, since $b = a^{-1} \mod \phi(n)$, then $ab \mod \Phi(n) = 1$ implies $ab = k\Phi(n)+1$ for some $k$. What we need to prove is that $x^{ab} \mod n = x$. In fact, if this holds we have:

$$D_{SK}(E_{PK}(x)) = (x^b)^a \mod n = x^{ab} \mod n = x$$

Notice now that $x^{ab} \mod n = x^{k\Phi(n)+1} \mod n = (x^{\Phi(n)})^k x \mod n$. Thus, in order to prove $x^{ab} \mod n = x$ it is enough to show that $(x^{\Phi(n)})^k x \mod n = x$. This proof is split into two cases:

- $gcd(x, n) = 1$. In this case we know that Euler theorem holds, and we directly have that $x^{\Phi(n)} \mod n = 1$ which implies $(x^{\Phi(n)})^k \mod n = 1$, giving the thesis;
- $gcd(x, n) > 1$. In this case, since $x < n$ we have that either $\gcd(x, n) = p$ or $\gcd(x, n) = q$. Without loss of generality we assume $\gcd(x, n) = p$ (the proof for the other case is identical). We now have $x = jp$ for some $j$ and $\gdc(x, q) = 1$. Euler theorem gives $x^{\Phi(q)} \mod q = 1$, which implies $x^{\Phi(q)\Phi(p)} \mod q = x^{\Phi(n)} \mod q = 1$. This implies $(x^{\Phi(n)})^k \mod q = 1$ giving $(x^{\Phi(n)})^k = wq+1$ for some $w$. We obtain
$$\begin{aligned} (x^{\Phi(n)})^k x \mod n &= (wq+1)x \mod n \\ &= (wq+1)jp \mod n \\ &= wqjp + x \mod n = x \text{ ,since } pq = n \text{ a} \end{aligned}$$

Thus, $D_{SK}(E_{PK}(x)) = x$ and the RSA cipher is correct.

**Example.** *We prove that* $x^{ab} \mod n = x$*, using* $p = 3$*,* $q = 11$ *and* $x = 2$*.*
*In this case we have that* $\phi(n) = (p-1)(q-1) = 20$*,* $PK = (b, n) = (3, 33)$ *and* $SK = (a, n) = (7, 33)$*. Thus:*

$$x^{ab} \mod n = x^{7*3} \mod 33 = x^{21} \mod 33 = 2^{21} \mod 33 = 2$$

### 3.7.4 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. There exists a much faster algorithm, known as **Square-and-Multiply**, that is based on the following observation: when we rise a number $x$ to a power of 2 such as 8, instead of performing 7 multiplications we can simply compute $((x^2)^2)^2 = xxxxxxxx$ which is just 3 multiplications. In general, if the exponent is not a power of 2, we can exploit a similar trick by performing some additional multiplications when needed. For example $x^{10}$ can be done as $((x^2)^2 x)^2 = xx\ xx\ x\ \ xx\ xx\ x$: after squaring twice we multiply the result by $x$ so that we have $x^5$. With a final square we obtain the result. In particular $x^{10}$ can be written as $(x^5)^2$ which is $(x^4 x)^2$ and finally $((x^2)^2)x)^2$: intuitively, if the exponent is even we divide it by 2 and we square, if it is odd we get one $x$ out (which is an additional multiplication) and we proceed as above. Dividing the exponent by 2 amounts to follow its binary representation.
For example, 10 is 1010. We start from the most significant bit. At each step we square and, only when we have a 1 we multiply by $x$. A python implementation follows:

```
def squareAndMultiply(x,e):
    r=1
    b = bin(e)[2:]      # binary representation of e, removes the 0b
    for bit in b:       # for all bits of exponent
        r = (r*r)               # we always square
        if bit=='1':
            r = (r*x)       # we multiply only if bit is 1
    return r
```

Figure 57: Square-and-Multiply algorithm

**Example.** *We compute* $2^{10}$*, where* $10 = 1010$ *and* $x = 2$.
*Initially, we have that* $r = 1$*, so in the first round we compute* $r = 1 * 1 = 1$*, the bit is 1, so we update* $r = r * x = 1 * 2 = 2$*. In the second round,* $r = 2 * 2 = 4$*, and the bit is 0, so we do not update it. In the third round,* $r = 4 * 4 = 16$ *and the bit is 1, thus* $r = 16 * 2 = 32$*. Finally, in the last round* $r = 32 * 32 = 1024$ *and the bit is 0, so the result is* $2^{10} = 1024$*, which is correct.*

The **number of steps** in the worst case is $O(k^2)$ for the two multiplications, iterated $k$ times, giving $O(k^3)$. Thus for 1024 bits we can expect about 1 billion steps, which is still efficient on modern machines.

**Inverse modulo** $\phi(n)$ This operation is necessary to compute the private exponent from the public one. Recall that RSA requires $ab \mod \Phi(n) = 1$. To find these two numbers, one technique is to choose $b$ and to compute its multiplicative inverse modulo $\Phi(n)$. This is guaranteed to exist, by the Euler theorem, when $\gcd(b, \Phi(n)) = 1$. In this particular case, there is also an efficient algorithm to compute the inverse based on the Euclidean algorithm for computing gcd (see the exercises).

### 3.7.5 Generating RSA exponents

We can thus pick a random **public exponent** $b$ and compute its **inverse** modulo $\Phi(n)$. This works if we are lucky and $b$ is coprime to $\Phi(n)$. This happens quite frequently. It can be proved that two random numbers are coprime with probability $\approx 0.6$. Thus iterating this 2-3 times should give a suitable pair of public, private exponents.
In practice, however, it is often the case that the **public exponent** is a **fixed constant**, typically the prime number $2^{16}+1 = 65537$. Using a low exponent improves the performance of encryption (recall that each 1 in the exponent requires an extra multiplication). In this case, of course, we have to choose $n$ so that $\Phi(n) = (p-1)(q-1)$ is not a multiple of 65537.

**RSA and factorization** Notice that knowing $\Phi(n)$ makes it very simple to compute the private exponent. Factoring $n$ into $pq$ allows for computing $\Phi(n) = (p-1)(q-1)$. Thus we now understand that RSA security is based on the **infeasibility** of **factoring** the modulus $n$. For this reason we require $n$ to be at least 1024 bits: numbers of this size are still considered impossible to factor in a reasonable time. A size of 2048 is however already suggested for critical applications. Moreover, if we compute $\Phi(n)$ we can easily factor $n$ since we have a system of 2 equations with two variables $p, q$:

$$\begin{cases} n = pq \\ \phi(n) = (p-1)(q-1) \end{cases}$$

By solving it we can find $p$ and $q$. We thus have that finding $\Phi(n)$ is at least as difficult as factoring. This gives **high confidence** of the **difficulty** of breaking the private exponent by simply computing the inverse modulo $\Phi(n)$. However, there could be other ways to break the cipher. We will discuss this more in detail in the next lessons.

### 3.7.6 Primality test

We finally see how to **generate two big prime numbers**. The oldest algorithm for finding prime numbers is due to Eratosthenes and is known as the **"Sieve of Eratosthenes"**. In order to discover all prime numbers less than or equal $N$ we write all numbers from 2 to $N$. We then start from the smallest (2, initially) that we call $p$ and we remove from the list all of its multiples $2p, 3p, ...$ We iterate the procedure by picking the smallest $p$ survived from previous steps.

**Example.** *Let $N = 10$, and we pick all the numbers from 2 to 10, i.e. $\{2,3,4,5,6,7,8,9,10\}$. Then, starting from 2, we remove 2,4,6,8,10, and we get $\{3,5,7,9\}$. Then, we take 3 and we remove 3 and 9, and we get $\{5,7\}$. Then, we take 5 and we remove 5, and we get $\{7\}$. Finally, we take 7 and we remove 7. Thus, the prime numbers between 2 and 10 are $\{2,3,5,7\}$.*

At the end, the **remaining numbers** are guaranteed to be **prime** since they are **not multiples of any smaller number**. The algorithm can be **optimized** by **removing multiples** starting from $p^2$. In fact, any smaller $np$ with $n < p$ would have been removed at a previous step when

checking the multiples of $n$. This also implies that we can stop when $p$ is the square root of $N$. A simple python implementation follows.

```
import sys, math

def Sieve(N):
    Ns = int(math.sqrt(N))              # square root of N
    sieve = set(range(2,N+1))           # sets are good to model the 'sieve'
    for p in range(2,Ns+1):
        if p in sieve:                  # for all survived numbers
            for q in range(p*p,N+1,p):  # we remove multiples bigger that p*p
                sieve.discard(q)
    print sieve                         # sieve have only primes
```

Figure 58: *Sieve of Eratosthenes* for primality test

Unfortunately this algorithm takes at least $N$ steps meaning $2^{1024}$ for the smallest RSA modulus. Instead of generating all the primes we pick them at **random** and **test their primality**. There are many **probabilistic efficient algorithms** for primality test. In 2002, Agrawal, Kayal and Saxena proposed the first deterministic algorithm for primality test in the paper *"PRIMES is in P"*[1] published on Annals of Mathematics. This is a very important result but the proposed algorithm is much slower with respect to existing probabilistic ones, as it costs $O(k^6)$ which makes it rather slow for $k > 1024$.

**Miller-Rabin test** Here we illustrate one of the most famous probabilistic **primality tests** due to Miller and Rabin. The algorithm is a **NO-biased Montecarlo**, meaning that it is always **correct** for the **NO** answer (the number is not prime) but can be wrong for the YES case (the number is prime). The probability of being wrong, however, is less than a constant $\epsilon = \frac{1}{4}$. This allows for **iterating** the test until the **error is small enough**: by running the test $r$ times, the probability that it says YES and the number is not prime is less than $\epsilon^r = \frac{1}{4^r}$, i.e., it **decreases exponentially** with respect to the **number of iterations**. For example if we pick $r = 128$ we have an error with probability less than $\frac{1}{4^{128}}$ which is less than brute-forcing a typical symmetric key, and so small to be negligible.
The idea is to exploit this test in order to assess if the random number chosen for RSA is prime or not.
Let us now illustrate the algorithm:

**Example.** *If $n = 7$, then we have*

- $n - 1 = 6 = 2^1 * 3$;
- $a = 2$;
- $b = 1$.

*Since $b = 1$, we stop, so we return TRUE.*

**Theorem.** *The Miller-Rabin algorithm is always correct on False answers, i.e. it is NO-biased.*

*Proof.* Assume, by contradiction, that the algorithm returns False but the number $n$ is prime. False is only returned at the last step which means the algorithm 'survived' all the if branches.

[1] https://www.cse.iitk.ac.in/users/manindra/algebra/primality$_v$6.*pdf*

```
Prime(n):
    write n-1 = 2^k * m
    pick a random a such that 1 < a < n
    b = a^m mod n
    if b == 1:
        return True
    for i in range(0,k):
        if b == n-1:
            return True
        b = b*b mod n
    return False
```

Figure 59: *Miller-Rabin* test

In particular we have that $a^m \mod n \neq 1$. Moreover $a^{2^i m} \mod n \neq n-1$, for $i = 0 \dots k-1$. By the assumption that $n$ is prime and by Euler theorem we know that $a^{\Phi(n)} \mod n = a^{n-1} \mod n = a^{2^k m} \mod n = 1$ (notice that $1 < a < n$ and by assumption $n$ is prime, so $a$ and $n$ are coprime). In summary we have:

$$\begin{array}{ccccccc} a^m \mod n & & a^{2^1 m} \mod n & & a^{2^2 m} \mod n & \dots & a^{2^{k-1} m} \mod n & & a^{2^k m} \mod n \\ \nparallel & & \nparallel & & \nparallel & & \nparallel & & \parallel \\ 1, n-1 & & n-1 & & n-1 & & n-1 & & 1 \end{array}$$

It is easy to show that, if $n$ is prime then the only square roots of 1 modulo $n$ are $n-1$ and 1. In fact, $r$ is a square root of 1 if and only if $r^2 \mod n = 1$, i.e., $r^2 = 1 + zn$ for some $z$, meaning that $r^2 - 1 = (r-1)(r+1) = zn$ which implies $\frac{(r-1)(r+1)}{n} = z$. Since $n$ is prime we have that $n$ must divide either $r-1$ or $r+1$ and so $r = -1, 1 \mod n$ (in this case $(r-1) = k_1 n$, so $r = k_1 n + 1$, or $(r+1) = k_2 n$ etc..). Notice that if $n$ is not prime there might be square roots of 1 different from 1 and $n-1$. For example for $n = 8$ we have that $5^2 \mod 8 = 1$ meaning that 5 is a square root of 1 modulo 8.
It is now sufficient to notice that each of the above values is the square of the preceding one. Since the last one ($a^{2^k m}$) is equal to 1 we have the the previous ($a^{2^{k-1} m}$) is a square root of 1 modulo $n$, i.e., it is 1 or $n-1$. Now we know that it is different from $n-1$ which implies it is 1. We iterate (backward) this on all the values until we get $a^m \mod n = 1$ which gives a contradiction. This proves that the algorithm can never return False if $n$ is prime.
A working python implementation follows. Notice the use of *squareAndMultiply* for exponentiation (this is necessary if we want to test large primes).

**Example.** *If $n = 5$, then we have that $m = 4$ and $k = 0$. We repeat two times the iteration of the while, resulting in $m = 1$ and $k = 2$, having $4 = 2^2 * 1$. We then choose $a = 2$, then $b = 2$: in the for loop we modify $b = 2 * 2 \mod 5 = 4$, so we return FALSE.*

### 3.7.7 Generating RSA primes

How can we finally generate big primes? We simply **generate** them at **random** and we **test** their **primality**. This takes about $\ln n$ steps, where $n$ is the upper bound of the interval we choose from. It can be proved, in fact, that about 1 out of $\ln n$ numbers less than n are prime.

```
def Prime(n):
    m = n-1
    k = 0
    while (m&1 == 0):       # while m is even
        m = m>>1
        k += 1
    a = random.randrange(2,n)
    b = squareAndMultiply(a,m,n)
    if b == 1:
        return True
    for i in range(k):
        if b == n-1:
            return True
        b = b*b % n
    return False
```

Figure 60: *Miller-Rabin* test - second version

For example, if we want 512 bits of size (for a modulus of 1024) we have that $\ln n = \log_2 n \times \ln 2 \approx 512 \times 0.7 \approx 358$. Thus, on average, after 358 attempt we should find a prime number of size 512 bits.
Notice that the **Miller-Rabin** algorithm should be **iterated** to lower as needed the probability of error in case of successful answer. Even if the original paper proved that such a probability is bounded by $\frac{1}{4}$, in a subsequent paper[2] it has been shown that the algorithm is much more precise than that. In practice this is the suggested number of iterations from NIST (Appendix C.3): 7,4 and 3 for 512, 1024 and 1536 bits, with an error probability of $2^{-100}$, meaning that the test requires very few iterations to give a prime with high probability.

### 3.7.8 Case study: openSSL

OpenSSL is a software library for applications that need to secure communications over the Internet widely used:

- Against eavesdropping;
- To be sure of the identity of the party at the other end.

It contains an open source implementation of the SSL and TLS protocols, and it supports a number of different cryptographic algorithms, such as AES, Blowfish, RSA, DSA etc..

## 3.8 Security of RSA

We now discuss the main results about **security** of RSA cipher.

[2]I. Damgard, P. Landrock, and C. Pomerance, C. "Average Case Error Estimates for the Strong Probable Prime Test" Mathematics of Computation, v. 61, No, 203, pp. 177-194, 1993.

### 3.8.1 Computing $\phi(n)$

As already discussed, security of RSA is based on the **secrecy** of $\phi(n) = (p-1)(q-1)$. It is however **trivial** to see that **computing** $\phi(n)$ is at least as **difficult** as **factoring** $n$. Given an algorithm that computes $\phi(n)$ it is enough to solve the system of equations:

$$\begin{cases} n = pq \\ \phi(n) = (p-1)(q-1) \end{cases}$$

to compute $p$ and $q$.

### 3.8.2 Computing the private exponent

It could be possible, in principle, to compute the **private exponent** without necessarily computing $\phi(n)$. It can be proved that this would allow to factor $n$:

**Theorem.** *Given an algorithm that computes exponent a we can write a probabilistic "Las Vegas" algorithm that factorises n with probability at least* $\frac{1}{2}$.

As for Monte Carlo algorithms, we can **iterate** a Las Vegas algorithm as needed. This proves that if an exponent is leaked then $n$ is compromised, thus breaking a private key is as difficult as factoring. Notice also that once we leak a private key the modulus is no more secure. This implies that $n$ should **never** be **reused** for different key pairs: any time we generate a key pair we need to generate a new modulus $n$.

### 3.8.3 Small encypryption exponent

We have already observed that implementations of RSA use small encryption exponents (typically $2^{16} + 1 = 65537$) to improve the performance of encryption function. We discuss some attacks that are possible if we choose an excessively small exponent such as 3.

**Theorem (Chinese Remainder Theorem).** *Let* $n_1, .., n_k$ *be integers* $> 1$, *and* $N = n_1 * .. * n_k$. *If the* $n_i$ *are pairwise coprime, and if* $a_1, .., a_k$ *are integers s.t.* $0 \leq a_i < n_i$ *for every* $i$, *then there is one and only one integer* $x$, *s.t.* $0 \leq x < N$ *and* $x \mod n_i = a_i$ *for every* $i$.

**Attack 1: Same message encrypted using different moduli** Suppose the same message $m$ is sent **encrypted** under at least **three different public keys** $(3, n_1), (3, n_2), (3, n_3)$ giving the three ciphertexts $c_1, c_2, c_3$ such that $c_i = m^3 \mod n_i$ (e.g. $c_1 = m^3 \mod n_1$, $c_2 = m^3 \mod n_2$ and $c_3 = m^3 \mod n_3$). Notice that **moduli** $n_1, n_2, n_3$ are very likely to be **coprime** as they will not share their prime factors. The Chinese Reminder Theorem applies and proves that there exists a unique $x < n_1 n_2 n_3$ such that $x \mod n_i = c_i$, with an efficient way to compute it (*Why do the assumptions of the Theorem hold?* Notice that in this case we have that $a_i = c_i$ and $x = m^3$). Notice now that $m < n_i$ which implies $m^3 < n_1 n_2 n_3$. By definition of $c_i$ we also have $m^3 \mod n_i = c_i$. Thus the unique $x$ given by the Chinese Reminder Theorem must be equal to $m^3$. It is now enough to compute $\sqrt[3]{x}$ to get $m$.

**Example.** *Let* $m = 2$, $n_1 = 3$, $n_2 = 5$ *and* $n_3 = 7$. *Then,* $n_1$, $n_2$ *and* $n_3$ *are pairwise coprime (notice also that* $PK_1 = (3, 3)$, $PK_2 = (3, 5)$ *and* $PK_3 = (3, 7)$*). We have that* $m^3 = 8 < n_1 * n_2 * n_3 = 105$ *and*

$$c_1 = m^3 \mod n_1 = 8 \mod 3 = 2$$

$$c_2 = m^3 \mod n_2 = 8 \mod 5 = 3$$

*and*

$$c_3 = m^3 \mod n_3 = 8 \mod 7 = 1$$

*Thus, it holds that* $0 \leq 2 < 3$, $0 \leq 3 < 5$ *and* $0 \leq 1 < 7$, *so there is one and only one integer* $x$ *s.t.* $0 \leq x < 105$ *and* $x \mod 3 = 2$, $x \mod 5 = 3$ *and* $x \mod 7 = 1$, *which is* $x = 8$. *This implies that* $m^3 = 8$, *from which we derive that* $m = 2$.

In general, $b$ encryptions of the same message $m$ under different keys are enough to recover $m$. Picking $b = 65537$ makes this attack quite unlikely.
Moreover the attack can be prevented by a **randomized padding** scheme such as *PKCS1* which transforms message $m$ into a $k$-bits long message $EM = 0x00||0x02||PS||0x00||m$, where $PS$ is a sequence of random bytes different from 0. In this way any time we encrypt $m$ we in fact encrypt a different plaintext. Notice that *PKCS1* enables padding-oracle attacks and is superseded by the more secure *Optimal Asymmetric Encryption Padding* (*OAEP*).

**Attack 2: Small messages** Padding is also needed because of the following trivial attack on **small messages** encrypted under **small exponents**. Consider a small $m$ encrypted with exponent $b = 3$ (we recall that the exponent $b$ is known by the attacker). We have $y = m^3 \mod n$. Now it might be the case that $m^3 \leq n$ meaning that $y = m^3 \mod n = m^3$. To decrypt is then enough to compute $\sqrt[3]{y}$. **Padding prevents** this **attack** by making the size of $m$ close to the size of the modulus $n$.
It could be possible that an attacker is able to **partially recover the plaintext**. Let $y = E_{PK}(x)$. Some examples of partial information follow:

$$\mathit{parity}(y) = \begin{cases} 0 & \text{if } x \text{ is even} \\ 1 & \text{otherwise} \end{cases}$$

$$\mathit{half}(y) = \begin{cases} 0 & \text{if } 0 \leq x < n/2 \\ 1 & \text{otherwise} \end{cases}$$

Function $\mathit{parity}(y)$ returns the parity of the plaintext, while $\mathit{half}(y)$ tells if the plaintext is less then half of the modulus.
First we show that the latter can be defined based on the former. This result exploits the following fundamental property of RSA.

**Theorem (RSA multiplicative property).** *RSA cipher is such that* $E_{PK}(x_1)E_{PK}(x_2) \mod n = E_{PK}(x_1 x_2 \mod n)$. *Intuitively, the multiplication of the encryption is equal to the encryption of the multiplication.*

*Proof.* It is enough to apply the definition of encryption:

$$\begin{aligned} E_{PK}(x_1)E_{PK}(x_2) \mod n &= x_1^b x_2^b \mod n \\ &= (x_1 x_2)^b \mod n = \\ &(x_1 x_2 \mod n)^b \mod n \\ &= E_{PK}(x_1 x_2 \mod n) \end{aligned}$$

We can now prove that $\mathit{half}(y) = \mathit{parity}(E_{PK}(2)\ y \mod n)$. Notice that $\mathit{parity}(E_{PK}(2)\ y \mod n) = \mathit{parity}(E_{PK}(2x \mod n))$. Consider now the case $\mathit{half}(y) = 0$. By definition we have $0 \leq x < n/2$ which implies $0 \leq 2x < n$, thus $2x \mod n = 2x$ which is certainly even, i.e., $\mathit{parity}(E_{PK}(2)\ y \mod n) = 0$. If, instead, $\mathit{half}(y) = 1$ then $n/2 < x < n$ which implies $n < 2x < 2n$ thus $2x \mod n = 2x - n$ which is certainly odd since $2x$ is even and $n$ is odd (it cannot be a multiple of 2). As a consequence $\mathit{parity}(E_{PK}(2)\ y \mod n) = 1$.
The following holds:

**Theorem.** *Given an algorithm that computes half$(y)$ or parity$(y)$ it is possible to compute the whole plaintext $x$.*

*Proof.* Since *half*$(y)$ can be defined in terms of *parity*$(y)$ (see above), it is sufficient to prove that having *half*$(y)$ we can compute $x$. It is enough to do a binary search, each time multiplying by $E_{PK}(2)$ the ciphertext, and getting the left/right interval depending on the value of *half*. It is obvious that this works for the first step (definition of half exactly tells us whether $x$ belongs to the first half or the second half of the interval). Then, think of *half*$(y\ E_{PK}(2))$. It is 0 when $2x \mod n \leq n/2$ which happens when $0 \leq x < n/4$ or $n/2 \leq x < 3n/4$ since we respectively obtain $0 \leq 2x < n/2$ and $n \leq x < 3n/2$, the latter implying $0 \leq x \mod n < n/2$. This reasoning can be applied, similarly, to next steps until a single solution survives. Since each time the interval in split in two, the number of steps is $\lceil \log_2(n) \rceil$.

**Chosen ciphertext attack** We conclude this section about security of RSA by illustrating a simple attack under the very powerful **attacker** that can **choose ciphertexts** to decrypt. The challenge is to decrypt a ciphertext $\tilde{y}$ by asking for decryptions of different ciphertexts $y_1 \dots y_n$. The attack proceeds as follows: we pick a random $r$ such that $1 < r < n$ and is invertible modulo $n$ (recall that the attacker sees $(b, n)$). The inverse can be computed using the extended Euclidean Algorithm (if it fails we pick another random $r$). We ask for decryption of $y_1 = \tilde{y}E_{PK}(r) \mod n$. We obtain $x_1 = \tilde{x}r \mod n$ where $\tilde{x}$ is the decryption of $\tilde{y}$ (exploiting the RSA multiplicative property). It is now sufficient to multiply this number by $r^{-1} \mod n$ to get $\tilde{x}$.

**Example.** *Let $n = pq = 3 * 7 = 21$, then the public key is $(17, 21)$, $y_1 = 1$ and $x_1 = 1$. Let us see if we find $\tilde{x} = 2$. We have that $ab \mod 12 = 1$, so $a * 17 \mod 12 = 1$, from which we derive that $a = 5$. Thus, the secret key is $(5, 21)$.*
*We then have to choose an $r$ s.t. $1 < r < 21$, in this case $r = 11$; thus:*

$$E_{PK}(11) = 11^{17} \mod 21 = 2$$

*and*

$$r * r^{-1} \mod 21 = 1$$

*, so*

$$11 * r^{-1} \mod 21 = 1$$

*, from which we derive that $r^{-1} = 2$. Let use assume we want to find $\tilde{x} = 2$, thus $\tilde{y} = 2^{17} \mod 21 = 11$. Then, $y_1 = \tilde{y}E_{PK}(r) \mod n = 11 * 2 \mod 21 = 1$, the attacker gets $x_1 = 1^5 \mod 21 = 1$, so $\tilde{x} = x_1 * r^{-1} \mod 21 = 1 * 2 = \mod 21 = 2$.*

## 3.9 Signatures, hashes and MACs

Asymmetric key cryptography is also employed to develop *digital signature schemes*, i.e., a digital counterpart of classic signature on paper. There are some important **features** of **classic signature** that deserve to be discussed in order to understand which are the basic expected properties of any good signature scheme.

1. Classic signature is **physically part** of the **signed document**. In a digital world this requires a mechanism the avoid that a signature can be just cut-and-pasted to any different document. To achieve this, we let the **signature depend** on the **signed document**: two different documents signed by the same entity will have different signatures;

2. Classic signature is **verified** by **comparing** it with an **official reference signature**. This is not possible in the digital world since we have just required (item above) that the signature of different documents should always be different. We thus need a **mechanism** to verify a signed document with respect to an entity (the signer);
3. Documents signed on paper **cannot be easily copied**. In the **digital world** we can trivially **cut-and-paste** bytes so any signed document can be replicated as many times as we need. In certain applications (e.g. e-commerce) signature needs to be integrated with mechanisms to avoid uncontrolled replicas.

The hypothesis are that:

- Everyone knows the *public key* of Bob;
- Only Bob knows his *private key*,

while the idea is that Bob sends a message signed using the digital signature. We want that:

1. To **sign**, you need to know the **private key**;
2. To **verify**, you need the **public key**.

### 3.9.1 Definition

A digital signature scheme consists of two **functions**:

- $Sig_{SK}(x)$ **generates** the **signature** of $x$ using a private key $SK$;
- $Ver_{PK}(x, y)$ **checks** the **validity** of **signature** $y$ on message $x$ using a public key $PK$. Returns true if the signature is valid and false otherwise.

Similarly to asymmetric cryptography, we require that these **two functions** are **easy** to **compute** knowing the keys. Moreover, it must hold $Ver_{PK}(x, Sig_{SK}(x))$.
For what concerns **security**, we require that **without** knowing the **private key** it is **computationally infeasible** to find a **message** $x$ and a **signature** $y$ such that $Ver_{PK}(x, y)$ holds, i.e., such that $y$ is a valid signature for $x$.

### 3.9.2 RSA-based digital signature (flawed first attempt)

We now discuss how the **RSA** cipher can be used to implement a **digital signature** scheme. We let

$$\begin{aligned} Sig_{SK}(x) &= D_{SK}(x) \\ Ver_{PK}(x, y) &= \begin{cases} true & \text{if } E_{PK}(y) = x \\ false & \text{otherwise} \end{cases} \end{aligned}$$

Notice that encryption and decryption of RSA are the same function. To sign we rise $x$ to the private exponent, modulo $n$, i.e.

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

, and to check the signature we rise the signature to the public exponent modulo $n$, and we check that the result is the same as the original message $x$.

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

Notice that is impossible to copy the key without knowing $SK$, while it is possible to check the signature knowing $PK$.

**Example.** *Let $p = 7$ and $q = 13$. Then the public key is given by $(5, 91)$, while the private key is given by $(29, 91)$, with $x = 2$. In this case*

$$Sig_{SK}(2) = D_{SK}(2) = 2^{29} \mod 91 = 32$$

*, and*

$$Ver_{PK}(2, 32) = \textit{true if } E_{PK}(32) = 32^5 \mod 91 = 2$$

**Problems and attacks** This scheme is **not** yet **satisfactory** as it does not satisfy the above security property. We show different ways to find $x$ and $y$ such that $Ver_{PK}(x, y)$.

- **Forging a "random" signed message**. We pick an arbitrary signature $y$ and we compute the corresponding signed message as $E_{PK}(y)$ since $Ver_{PK}(E_{PK}(y), y) = \text{true}$, by definition. Of course the signed message will be meaningless but still it is a valid forged signature and depending on the application it could be accepted as valid (e.g., when the expected message only contains a number);
- **Multiplying signed messages**. If we have two signed messages $x_1$, $x_2$ with signatures $y_1$ and $y_2$, then $y_1 y_2 \mod n$ is the signature of a message $x_1 x_2 \mod n$. Again, if the expected message is just a number with no particular format or padding, this attack might be very effective.

In addition to the discussed forging attacks, the above defined signature has different **drawbacks**:

- The **size** of the **signature** is at **least** the same as the size of the **message** meaning that we have to send at least double the size of signed data;
- If a **message** is **bigger** than the **RSA modulus** we would need to **split** the message into **blocks** and use some encryption mode to encrypt the different blocks. Recall, however, that none of the encryption modes we have discussed for symmetric ciphers provide a satisfactory level of integrity. This would imply possible attacks in which both the message and the signature are manipulated, making the system not able to ensure *integrity* (i.e. that the information can be modified only by an authorized user);
- **Asymmetric** cryptography is much **slower** than **symmetric** one. Even adopting variants of encryption modes with strong integrity properties we would need to perform many encryptions to sign a message, and this could become unaffordable for long files.

The **solution** to all of these issues, including the problem of forging, is the adoption of **cryptographic hash functions**. A hash function $h : X \to Z$ is a function taking an arbitrarily long message $x$ and giving a digest $z$ of fixed length. When used in cryptography, these functions are required to satisfy **specific properties** that we discuss below through our running-example.

### 3.9.3 RSA-based digital signature (hash-based)

We modify our proposed signature scheme based on RSA so to prevent all of the discussed problems. Let $h$ be a hash function.

$$\begin{aligned} Sig^h_{SK}(x) &= D_{SK}(h(x)) \\ Ver^h_{PK}(x,y) &= \begin{cases} true & \text{if } E_{PK}(y) = h(x) \\ false & \text{otherwise} \end{cases} \end{aligned}$$

Thus, to **sign** $x$ we **decrypt** its hash under the **private key**. To **check** the signature we **encrypt** the signature under the **public key**, we recompute the hash on $x$ and we compare the two results.
Since the **hash** is of fixed length, it is enough to take it **smaller** than the **RSA modulus** to solve all the issues related to the size of the signature and the necessity of encryption modes: a signature would always be as long as the RSA modulus and will always need just one RSA encryption (plus the hash computation).
**Forgery** is more delicate. Consider the simplest attack (in the scheme with no hash) where, given an arbitrary signature $y$ we compute the corresponding signed message as $E_{PK}(y)$ since $Ver_{PK}(E_{PK}(y), y) = \text{true}$, by definition. In the hash-based scheme, $E_{PK}(y)$ would provide the hash of the signed message, so if we are able to find $x$ such that $h(x) = E_{PK}(y)$ we will have that $Ver^h_{PK}(x, y) = \text{true}$. This leads to our **first property** for cryptographic hash functions.

**Definition (Preimage resistance).** *A hash function $h$ is **preimage resistant** (or **one-way**) if given $z$ it is infeasible to compute $x$ such that $h(x) = z$.*

**Example.** *We consider a simple hash function $h(x)$ that splits message $x$ into blocks $x_1, x_2, \ldots, x_n$ of a fixed size $k$ and computes $h(x) = x_1 \oplus x_2 \oplus \ldots \oplus x_n$, i.e., the bit-wise xor of all blocks. This hash function is clearly not preimage resistant: given a digest $z$ we can easily find preimages. For example, $z$, $z||0$, $z||x||x$, $z||x||y||x \oplus y, \ldots$ where 0 notes the block of $k$ zeros and $||$ notes the concatenation of blocks, are all correct preimages of $z$.*

Notice that **preimage resistance** also **prevents forging** based on RSA **multiplicative property**. If we have two signed messages $x_1$, $x_2$ with signatures $y_1$ and $y_2$, we have that $y_1 y_2 \mod n$ is the signature of a message whose hash is the same as $z = h(x_1)h(x_2) \mod n$. Finding such a message $x$ means that we can compute a preimage $x$ of $h$ such that $h(x) = z$, but this is ruled out by preimage resistance.
Thus, it appears that adopting a **hash-based RSA** signature scheme in which the hash function is **preimage resistant** solves all the issues. Unfortunately, we also have to consider **potential problems** deriving from the adoption of **hashes**: since a hash summarises messages as fixed-length digests, it can of course occur that **different messages have the same hash** (this phenomenon is addressed as **collision**). For example we might have different $x_1$ and $x_2$ such that $h(x_1) = h(x_2)$. Then, $Sig^h_{SK}(x_1) = Sig^h_{SK}(x_2)$, i.e., the two messages have the same signature.
Consider the following attack scenario: given a message $x_1$ and its signature $y_1$, the attacker computes a $x_2$ such that $h(x_1) = h(x_2)$. Then, $y_1$ is a valid signature for $x_2$ meaning that the attacker has forged a signature for a message of his choice. To prevent this problem we require the following property.

**Definition (Second-preimage resistance).** *A hash function is **second-preimage resistant** if given $x_1$ it is infeasible to compute $x_2$ such that $h(x_1) = h(x_2)$.*

If we assume the attacker is allowed to ask for signatures (similarly to what happens in a chosen-plaintext attack) it might still happen that he chooses **two different messages** $x_1$ and $x_2$ with the **same hash** and asks for a **signature** on $x_1$. In this way he obtains a valid signature for $x_2$. To exemplify, suppose he manage to generate two messages that look the same but one ($x_2$) gives

more advantage than the other ($x_1$), such as in two contracts with different prices. The attacker convinces the other party to sign $x_1$ which looks reasonable but obtains a signature on $x_2$ that has an outrageous price in it. For this reason, the ultimate ideal property for hash function is as follows.

**Definition (Collision resistance).** *A hash function is **collision resistant** if it is infeasible to compute different $x_1$ and $x_2$ such that $h(x_1) = h(x_2)$.*

Differently from *second-preimage resistance*, here the attacker can **choose** both $x_1$ and $x_2$ and he is not given a $x_1$ that he has to find a second-preimage of. Thus this **latter** property **implies** the **previous** one, i.e., if a **hash** is **collision resistant** it is also **second-preimage resistant**. It is possible to prove that **collision resistance** also **implies preimage resistance**. Thus, if a hash function is collision resistant it has all of the above mentioned properties. This result holds under the assumption that the number of messages we can hash is at least twice as the number of digests.

**Theorem.** *Let $h : X \to Z$ be a **collision resistant** hash function such that $|X| \geq 2|Z|$. Then $h$ is also **preimage resistant**.*

**Proof.** We prove the following equivalent fact: if $h$ is not preimage resistant then $h$ is not collision resistant. To do so, we show that given an algorithm $Invert(z)$ for inverting $h$ (that breaks preimage resistance) we can write a Las Vegas probabilistic algorithm that finds a collision. The algorithm is, in fact, rather simple: we pick a random message, we compute its hash and we invert it using the given algorithm $Invert(z)$. If we find a different message we are done otherwise, if we are unlucky and get the initial message, we FAIL (i.e. the function will not be collision resistant).
The **algorithm** is shown below.

```
FindCollisions:
    choose a random x1 in X
    z = h(x1)
    x2 = Invert(z)
    if x1 != x2:
        output(x1,x2)
    else:
        FAIL
```

Notice that the **correctness** of the solution is obvious since $x_1$ and $x_2$ have the **same hash** meaning that they are a **collision**. We now prove that failure happens with probability at most 1/2. Since we have $|Z|$ possible digests, we have that $Invert(z)$ returns exactly $|Z|$ preimages, one for each digest, and these are the 'unlucky' cases, i.e., the messages that once hashed will be mapped back to themselves. Thus the good cases are exactly $|X| - |Z|$ and the probability of success is $\frac{|X|-|Z|}{|X|} \geq \frac{|X|-\frac{1}{2}|X|}{|X|} = \frac{\frac{1}{2}|X|}{|X|} = \frac{1}{2}$ (since $|X| \geq 2|Z|$, then $|Z| \leq \frac{1}{2}|X|$). As a consequence we fail with probability $\leq \frac{1}{2}$. As any Las Vegas algorithm, this can be iterated as needed. For $r$ iterations we have that the probability of failure is $\leq \frac{1}{2^r}$.

**Example.** *Hash $h(x)$ of previous example is clearly not collision and second preimage resistant. Given $x = x_1, x_2, \ldots, x_n$ we have, for example, that $x_2, x_1, \ldots, x_n$ has the same digest. We can swap blocks (xor is commutative). We can add zero blocks. We can add whatever block twice or add it once and then add it xored with the original message, and so on.*

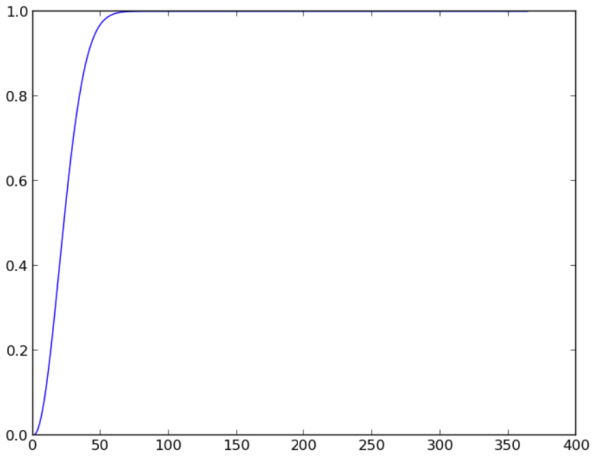
**Example.** *We consider another simple hash function $g(x) = E_{h(x)}(0)$, i.e., we use the previous hash to obtain a key for a cipher and we encrypt the constant 0 under that key. This hash function is preimage resistant if the adopted cipher is resistant to known plaintext attacks. In fact, finding $h(x)$ from $g(x)$ corresponds to breaking the cipher knowing the plaintext 0 and the ciphertext $g(x)$. However, it is neither collision resistant nor second preimage resistant, as collisions on $h$ are collisions on $g$ and we have shown above that $h$ is not collision resistant.*

**Birthday attack** We have discussed important properties of hash functions that are needed, for example, when developing a hash based signature scheme. Collision resistance is the strongest of such properties: it is important that hash functions are carefully designed so to make the computation of collisions infeasible. As for cryptography, however, the **attacker** can try to **break** a **function** by **brute force**. For the case of hash function, brute forcing is much simpler than expected thanks to the so called *Birthday Attack*.
This funny name comes from an analogy to the famous birthday paradox: given a group of only 23 people with probability 1/2 there are at least two with the same birthday (day and month, not year). With a group of 41 the probability is already around 90%. This true fact is so counter-intuitive to be called paradox.
The analogy with cryptographic hash functions is immediate: birthday can be seen as a hash function from any person to a fixed-size set of 365 days of the year. Two people having the same birthday represent a collision on the hash function. The fact collisions are so likely means that brute-forcing a hash function might be much easier than expected, and we will give a precise estimate of this.
Let us now show where the *'paradox'* comes from. We assume **people** being **mapped** to birthdays in a **uniform** way. This assumption fits very well with cryptographic hashes as they usually behaves that way to make finding a collision or a preimage as much hard as possible. We now compute the **probability** that in a group of $k$ people **none** share the **birthday**. For the first person any birthday is fine. Thus we have probability 1 of success. The second person should not share the birthday with the first one, meaning that we have probability 364/365 of success, and so on. We require that these events all occur together giving a probability of $\prod_{i=0}^{k-1} \frac{365-i}{365}$. Consequently, the probability of a collision is $1 - \prod_{i=0}^{k-1} \frac{365-i}{365}$. The following code computes this probability.

```
from decimal import *
def Birthday(k):
        r = 1
        for i in range(0,k):
            r=r*(365-i)
        return 1 - (Decimal(r)/Decimal(365**k))
```

Plotting over all values from 1 to 365 shows how fast the probability grows.
It is now useful to find a **relation** between the **number giving probability 1/2** (in this case 23) and the **number of digests**, as this will allow us to **estimate** the **cost** of a **brute force attack**. We let $n$ be the number of digests (365 for the birthday paradox). We reason as follows: the probability that we do **NOT** find a collision is

$$\prod_{i=0}^{k-1} \frac{n-i}{n} = \prod_{i=0}^{k-1} \left(1 - \frac{i}{n}\right)$$




Now, from analysis we know that, for small $x$, $1 + x \approx e^x$ (from Taylor series $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$). Thus, the probability of finding a collision is

$$\epsilon \approx 1 - \prod_{i=0}^{k-1} e^{-\frac{i}{n}} = 1 - e^{-\frac{(k-1)k}{2 \cdot n}}$$

Thus $1 - \epsilon \approx e^{-\frac{(k-1)k}{2n}}$. We thus get $ln(1-\epsilon) \approx -\frac{(k-1)k}{2n}$ . Thus $2n \cdot ln(1/(1-\epsilon)) \approx k^2 - k$. By a further approximation (disregarding $k$) we get

$$k \approx \sqrt{2n \cdot ln(1/(1-\epsilon))}$$

For $\epsilon = 1/2$ this gives $k \approx 1.17\sqrt{n}$. Thus, a brute-force attack on a hash functions with $n$ digests finds a collision with probability $1/2$ after about $\sqrt{n}$ attempts. For example, if a hash function returns digests of 128 bits it takes about $\sqrt{2^{128}} = 2^{64}$ trials to break it. So to have the same security we have with a 128 bit cipher we need double the length, i.e., 256 bits of hash.

### 3.9.4 Commonly used hash functions

One of the most famous cryptographic hash function is **MD5** (Message Digest 5), by Ron Rivest (the 'R' of RSA). It is a 128-bit hash used in many applications. Recently it has been shown to be **not collision resistant**. Moreover the relatively small size allows for a birthday attack in 'only' $2^{64}$ steps.
The most common alternative to MD5 is **SHA** (Secure Hash Algorithm). The second generation SHA (**SHA-2**) has a variable digest size from 224 to 512 bits. It is not vulnerable to the collision attacks of RSA and the length makes birthday attack infeasible.
The National Institute of Technology (NIST) has just concluded the selection of SHA-3, with a process similar to the one for selection AES. The winner is Keccak.

### 3.9.5 Message Authentication Codes (MACs)

We conclude by illustrating cryptographic **mechanisms** to achieve **authentication** using **symmetric keys**. Message Authentication Codes (**MACs**) are hash functions with a symmetric key. They produce a **fixed-size digest** of a message, whose value depends on the given key. The property we require is similar to what we asked for signatures: **without knowing the**

**key** $k$ it should be computationally **infeasible** to find a **message** $x$ and a MAC $y$ such that $MAC_k(x) = y$, i.e., such that $y$ is the MAC for $x$ under key $k$.
The MAC is checked by recomputing it and comparing with the received one. For example consider:

$$A \xrightarrow{x, MAC_k(x)} B$$

After receiving the message, Bob recomputes $MAC_k(x)$ and **compares** the result with the received MAC. If they matches, he can conclude the message comes from Alice.
Now the questions are:

1. How can we **implement** a MAC?
2. How can we **create** the digest?

We start by answering to the first question.

**CBC-based MAC** We recall the encryption and decryption schemes of CBC to be the following:

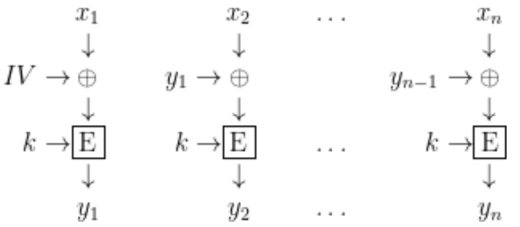



Figure 61: CBC: Encryption

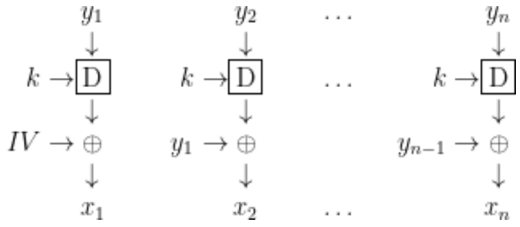



Figure 62: CBC: Decryption

A simple example of how to implement a MAC is by using CBC encryption mode (with a zero IV).
We then define $MAC_k(x) = y_n$, i.e., we take the **last encrypted block as digest**. Intuitively, since this block **depends** to all the **previous ones** (thanks to the chaining) the last block summarizes all the content of message $x$ and it clearly depends on the key $k$. The fact we use a cipher should also make it **hard** to **forge** a MAC without knowing the key. Finally, notice how from a message of $n$ blocks $(x_1, x_2, .., x_n)$, the result is a single block $y_n$.
However, in practice, the proposed MAC is **not satisfactory** as illustrated by the following simple **chosen-text forgery**: suppose to have a 1-block message $x$ and the relative $MAC_k(x)$. We have that $MAC_k(MAC_k(x)) = E_k(E_k(x)) = MAC_k(x||0)$ (notice that $MAC_k(x) = E_k(x)$

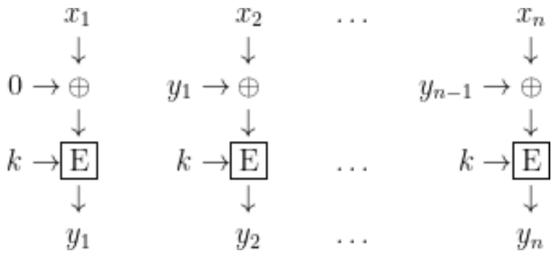
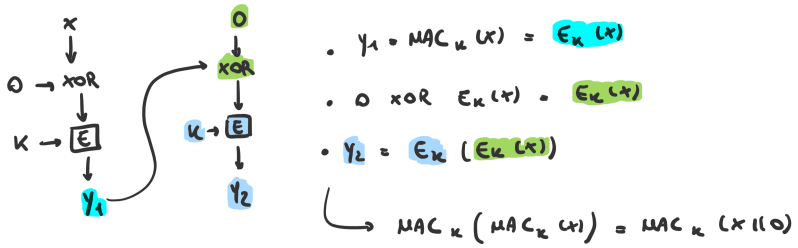



Figure 63: CBC-based MAC: Encryption

holds since $x$ is composed by only 1 block). Thus, if the attacker asks (chosen-text attack) for $MAC(MAC_k(x))$ he obtains a MAC for the two-blocks message $x||0$.

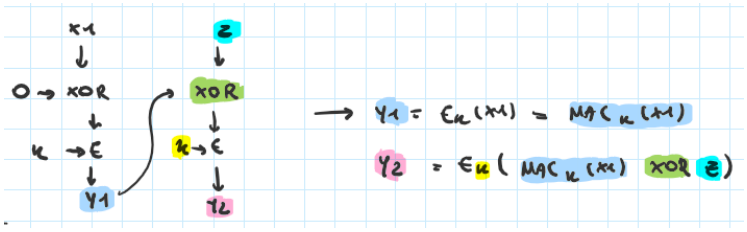



The following less-trivial forgery allows for more control on the attacked message. Consider two 1-block messages with their MACs $x_1$, $MAC_k(x_1)$ and $x_2$, $MAC_k(x_2)$. Let $H_1 = MAC_k(x_1)$ and $H_2 = MAC_k(x_1)$. Then

$$\begin{aligned} MAC_k(x_1||z) &= E_k(H_1 \oplus z) \\ &= E_k(H_1 \oplus H_2 \oplus z \oplus H_2) \\ &= MAC_k(x_2||H_1 \oplus H_2 \oplus z) \end{aligned}$$

Some notes:

- $MAC_k(x_1||z) = E_k(H_1 \oplus z)$ comes from the scheme below;




- $E_k(H_1 \oplus z) = E_k(H_1 \oplus H_2 \oplus z \oplus H_2)$ comes from the fact that if we XOR twice the same quantity, we obtain the same result;
- Finally, the last equivalence can be obtained with a scheme similar to the previous ones.

Thus, if the attacker wants to extend message $x_1$ with an arbitrary second block $z$, he can ask for $MAC(x_2||H_1 \oplus H_2 \oplus z)$. The obtained MAC will be valid for message $x_1||z$.
To **prevent** these **forgeries**, the ISO/IEC 9797-1 standard adds **additional transformations** to the **final encrypted CBC block**. For example, given an additional key $k'$ we define

$MAC_k(x) = E_k(D_{k'}(y_n))$. Changing the last step is a way to **avoid** that **MACs** can be **forged** from **MACs of shorter messages**, as done above. As an exercise, you can check that the above attacks do not work under this variation.

**Hash-based MACs** Another way to implement MACs is to base them on cryptographic hash functions. The most famous is called HMAC, by Mihir Bellare, Ran Canetti, and Hugo Krawczyk (see also RFC 2104).
The idea is to iterate a given hash function $h$ over blocks of $B$ bytes. We typically take $B$ as 64 bytes, i.e., 512 bits. We let:

- $ipad$ = the byte 0x36 repeated $B$ times;
- $opad$ = the byte 0x5C repeated $B$ times

$$HMAC_k(x) = h(k_p \oplus opad, h(k_p \oplus ipad, x))$$

,where $k_p$ is obtained from $k$ by appending 0 up to the byte length $B$. Notice that in this case we have **authentication** and **integrity**, but we do **not** have **non-repudiation**, since we both have the same key.
Interestingly, the authors show that if an attacker is able to forge HMAC that he is also able to find collisions on the underlying hash function (even when fed with a random secret as done here). Thus, if the hash function is collision resistant than HMAC is unforgeable.

**MACs vs. signatures** What is the **difference** between **MACs** and **signatures** given that they seem to provide very similar guarantees? To understand this crucial issue think of Alice sending to Bob a contract $x$. To prove **authenticity** (the message comes from Alice) she can decide to either sign it as $Sig_{SK}(x)$ with her private key SK or compute a MAC as $MAC_k(x)$ under a key $k$ shared with Bob. After receiving the message, Bob checks the signature or recomputes the MAC to verify its authenticity. What now if Alice denies to have ever sent $x$ to Bob? In other words, has Bob a way to show to to a third party (a judge) that the contract is from Alice? Here it becomes crucial the **symmetry** of the key: Alice is the only one knowing her private key $SK$ while both Alice and Bob know the shared key $k$. It is clear that, while signature can only be generated by Alice, a MAC can be easily 'forged' by Bob.
This important difference can be summarized by stating that **MACs never** provide **non-repudiability** and, more specifically, they do **not allow** to prove **authenticity** to a third party.

## 3.10 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;
12. Perform encryption and decryption using RSA cipher for the following: $p = 17$, $q = 2$, $b = 3$, $x = 2$. Use Euler’s theorem to compute $a$. Solution on slide 14-15 of L15;
13. Try to compute $2^{13}$ where $13 = 1101$ and $x = 2$. Solution on slide 23 of L15;
14. Try to efficiently compute $3^7$. Solution on slide 25 of L16;
15. Apply the "Sieve of Eratosthenes" algorithm for $N = 15$. Solution on slide 19 of L16;
16. Apply the "Sieve of Eratosthenes" algorithm for $N = 22$. Solution on slide 21 of L16;
17. Apply the "Sieve of Eratosthenes" algorithm for $N = 32$;
18. Apply the "Sieve of Eratosthenes" algorithm for $N = 18$. Solution on slide 27 of L16;
19. Apply the "Sieve of Eratosthenes" algorithm for $N = 17$. Solution on slide 3 of L17;
20. Apply the "Sieve of Eratosthenes" algorithm for $N = 21$;
21. Run the *Miller-Rabin* algorithm for $n = 10$;

22. Run the *Miller-Rabin* algorithm for $n = 9$. Solution on slide 11 of L17;
23. Run the *Miller-Rabin* algorithm for $n = 5$. Solution on slide 18 of L17;
24. Run the *Miller-Rabin* algorithm for $n = 15$. Solution on slide 29 of L17;
25. Run the *Miller-Rabin* algorithm for $n = 21$;
26. Perform encryption and decryption using RSA cipher for the following: $p = 13$, $q = 2$, $a = 5$, $x = 2$. Use Euler's theorem to compute $b$. Solution on slide 33-34 of L17;
27. Prove that $parity(y) = half(E_{PK}(2^{-1})y \mod n)$.

# 4 Applied cryptography

## 4.1 Strong authentication based on symmetric-key cryptography

We have seen that **passwords** and **PINs** suffer from **interception** and **replay**: an attacker sniffing a password can reuse it arbitrarily to authenticate. This can be improved using **OTPs**, i.e., passwords that are never reused. But even in this case, if the attacker is in the middle, he can sniff the password in transit and use it to authenticate once.
The problem with passwords and PIN is that we prove their knowledge by exhibiting the secret value. **Strong authentication** techniques, instead, allow for **proving** the **knowledge** of a secret **without showing it**. This can be achieved by showing a value that depends on the secret but does not allow to compute it.

### 4.1.1 Symmetric key authentication protocols

We discuss **strong authentication** protocols based on **symmetric-key cryptography**. The **secret** shared among the claimant and the verifier is a **symmetric cryptographic key**. In order to prove the knowledge of the key $K$, and thus her identity, the claimant sends to the verifier a message encrypted under $K$. Since generating an encrypted message without knowing the key is assumed to be infeasible, this proves its knowledge.
The general **idea** is to have a **challenge** and a **response**:

- **Challenge**: the verifier challenges the claimant to send a particular message encrypted under $K$;
- **Response**: the claimant sends the required message.

For example, a challenge might be "send me your name encrypted under $K$". The response from Alice would then be $E_K(A)$. The verifier will decrypt the message and will check that it matches name A. However, even if this does not reveal $K$, if the challenge is always the same an attacker can simple intercept $E_K(A)$ and replay it to authenticate as Alice. We thus have that the **challenge** should **never be the same**. A way to achieve this is to add a **time-variant parameter**.

**Sequence numbers** The challenge becomes "send me your name and your sequence number encrypted under $K$" with response $E_K(A, seq_A)$. We note the protocol as:

$$A \to B : E_K(A, seq_A)$$

, i.e. "A sending to B the encryption, under the key $K$, of A and $seq_A$.
The sequence number of Alice is initialized to 0, and then increased by 1 every time so that it never repeats. In a sense, we are following the **same idea** behind **OTPs**: we **never** send the **same response** twice so that it cannot be replayed. The verifier has to store the last sequence number from Alice so that he can decrypt the message and check that both A and the sequence number (increased by 1) match. In this case he accepts Alice identity and increments the sequence number so that it is ready for next authentication.
Sequence numbers have the **drawback** of requiring the verifier to store the last sequence number of each possible claimant. Moreover it is **unclear** what to do if for any reason (system or network failure) the **sequence numbers go out of sync**: restarting from 0 is unacceptable since any old authentication would become reusable by an attacker. An authenticated protocol to resync is necessary but it cannot be based on sequence numbers, of course.

**Timestamps** The challenge is "send me your name and a recent timestamp encrypted under K". So the protocol is:

$$A \to B : E_K(A, t_A)$$

, where the timestamp $t_A$ is the time (integer number) at the local clock of Alice when sending the message. The verifier decrypts the message and check that Alice name matches. Then he extracts a local timestamp $t_B$ and verifies that $t_B - t_A < W$ where $W$ is the acceptance-window, i.e., the maximum allowed delay for the received message. For example, if $W$ is 1 minute the timestamp of A have to be at most 1 minute behind the timestamp of B.
In order to **prevent replays** we should now pick $W$ so that it is **big enough** to receive honest messages but so small that no replay will ever be accepted. This is very **hard** in practice since delays on networks can vary a lot. For this reason, $W$ is typically taken **much bigger** than the **average delivery time**. To avoid replays, all the received timestamps are buffered so that double-reception of a valid timestamp can be easily checked. Periodically, the expired timestamps (out of $W$) in the buffer are deleted.
For example consider the following message, intercepted by the attacker:

$$A \to B : E_K(A, t_A)$$

Bob accepts Alice identity and stores $t_A$ in the buffer. The attacker $E$ tries a replay still inside $W$ (we write $E(A)$ to note the attacker pretending to be Alice):

$$E(A) \to B : E_K(A, t_A)$$

The timestamp is still valid but Bob finds it in the buffer and refuses authentication. Later on, when $t_A$ expires it is deleted from the buffer. Any further attempt of replay from the attacker will be refused since $t_A$ is out of $W$.
Timestamps do not require to store any per-user information (such as sequence numbers). It is enough to termporarily buffer the received-and-still-valid timestamps. Moreover, in case synchrony is lost it is enough to synchronize local clocks. It is however important to notice that this **synchronization** should be **authenticated** as **malicious** changes in clocks might easily allow the attacker to make **old timestamps valid** or to prevent any honest message to be accepted. As for sequence numbers, this authenticated synchronization cannot be implemented based on timestamps.

**Nonces** We have seen that sequence numbers and timestamps are based on some form of authenticated synchronization. We thus need at least one time variant parameter that do not assume any form of synchronization and can be used, if needed, to synchronize sequence numbers and clocks.
Nonces are "Numbers used only once", and here the challenge implies an **additional message** from the **verifier** to the claimant containing the nonce (notice that in this case we have a 2-steps protocol, while the previous ones were 1-step protocols). The challenge is "send me your name and nonce $N_B$ encrypted under key $K$".

$$B \to A : N_B$$
$$A \to B : E_K(A, N_B)$$

Bob generates a random nonce $N_B$ and sends it to Alice. He then decrypts the received message and checks that both A and $N_B$ match. In this case he accepts Alice identity. The nonce is discarded as it is supposed to be used only once.

If nonces are **big enough** (e.g. 128 bits), picking them at **random** implies that the **probability** of **reusing** the same nonce is **negligible**. Thus any replay will be prevented since the nonce will mismatch with overwhelming probability. An example follows:

$$B \to E(A) : N'_B$$
$$E(A) \to B : E_K(A, N_B)$$

Bob refuses since $N_B \neq N'_B$.

**ISO/IEC 9798-2 protocols** Protocols from the standard ISO/IEC 9798-2 are exactly as the ones discussed above apart that they enclose the identifier of the verifier B (to prevent reflection) instead of A.
**One-pass unilateral authentication**, i.e. A wants to authenticate with B

$$A \to B : E_K(ts_A, B)$$

,where $ts_A$ is a timestamp or a sequence number (then B will check is $ts_B - ts_A < W$).
**Two-pass unilateral authentication**

$$B \to A : N_B$$
$$A \to B : E_K(N_B, B)$$

**Two-pass mutual authentication**: here Alice and Bob authenticate each other.

$$A \to B : E_K(ts_A, B)$$
$$B \to A : E_K(ts_B, A)$$

where $ts_A$, $ts_B$ are either timestamps or sequence numbers. Notice that this is just the composition of two independent unilateral authentication protocols.
**Three-pass mutual authentication**

$$B \to A : N_B$$
$$A \to B : E_K(N_A, N_B, B)$$
$$B \to A : E_K(N_B, N_A)$$

This protocol can be understood starting from the composition of two unilateral authentications based on nonces:

$$B \to A : N_B$$
$$A \to B : N_A, E_K(N_B, B)$$
$$B \to A : E_K(N_A, A)$$

Now, including the nonce $N_A$ in the encryption of the second message is harmless and makes the two unilateral authentications tied in a unique mutual authentication session. The same holds for adding $N_B$ in the third message. Moreover, the fact that intended receiver (Bob) is specified in the second message together with challenge $N_A$ (that is now encrypted) makes it possible to remove A from the last message, as it is enough to prevent reflections.

### 4.1.2 Attacks on cryptography

Since challenge-response protocols provide cryptographic material to the attacker it is important to avoid as much as possible scenarios that facilitate cryptanalysis. For example, the protocol:

$$A \to B : E_K(A, t_A)$$

is likely to provide a **known-plaintext scenario**, since it is not hard to guess the value of Alice time-stamp, with some approximation.
More critically, the protocol:

$$B \to A : N_B$$
$$A \to B : E_K(A, N_B)$$

provides a **(partial) chosen-plaintext scenario** where the attacker can ask for encryption of any plaintext $A, z$ with arbitrary $z$. In fact it in enough for the attacker to impersonate Bob, noted $E(B)$, and send $z$ as nonce:

$$E(B) \to A : z$$
$$A \to E(B) : E_K(A, z)$$

Alice becomes a sort of "encryption oracle".
To **avoid** these **problems** a typical **countermeasure** is to **randomize cryptography**. This can be done in different way. For example, by adding a **random padding** to the plaintext. At a logical level, we can think of appending a random number $R_A$ that we call "confounder", as follows:

$$A \to B : E_K(A, t_A, R_A)$$

and

$$B \to A : N_B$$
$$A \to B : E_K(A, N_B, R_A)$$

In this way, the known and chosen-plaintext scenario are prevented. When decrypting, the random confounder will be ignored by Bob. We will always assume this form of randomization at the cryptographic level, with no need of making it explicit at the protocol level.

**Redundancy** Consider the following protocol based on sequence numbers (but it also works with timestamps):

$$A \to B : A, E_K(seq_A)$$

The **identifier** A is sent in the **clear** while the **sequence number** in **encrypted**. Assume that Bob only checks monotonicity of $seq_A$ (i.e. checks if $seq_A >$ last sequence) so to deal more flexibly with network delays (even if some message is lost the next will be accepted as the sequence number will be bigger than the stored one). In this case, if the attacker sends an arbitrary message:

$$E(A) \to B : A, z$$

the probability that the decryption of $z$ is a valid number bigger than the stored one is probably very high (especially if we start from small sequence numbers).
Encrypting arbitrary numbers with no format or message redundancy makes it impossible to check the integrity of the decryption: a random $z$ can always be decrypted in a meaningful arbitrary number. The presence of the identifier mitigates this problem since $z$, once decrypted, should at least match A. If A is $n$ bits long the probability that this happen is $1/2^n$.
As for randomization, there are **standard techniques** to **add redundancy**: a simple one is to enclose a **one-way hash** of the encrypted message, called **witness**, as in $h(seq_A), E_K(seq_A)$. The hash is a **proof** that the **sender knows** the **content** of the **message**. When Bob decrypts he checks that the hash of the decrypted message matches the received one. The attacker might send arbitrary $w, z$, but with a hash of 128 bits the probability of passing the test would be $1/2^{128}$, thus negligible. The fact the hash is one-way makes this technique applicable even when it is important to preserve the secrecy of the sent message.

**Reflection** We now discuss another reason why it is **important** to have the **identifier** encrypted together with a **time variant parameter**. Consider the protocol

$$A \to B : A, E_K(t_A)$$

Suppose that Bob is allowed to run the same protocol to authenticate with Alice using the same shared key $K$:

$$B \to A : B, E_K(t_B)$$

The attacker can pretend to be Bob, written E(B), as follows:

$$A \to E(B) : A, E_K(t_A)$$
$$E(B) \to A : B, E_K(t_A)$$

The message from Alice trying to authenticate with Bob is sent back (reflection) to Alice in a second session where the attacker pretends to be Bob. If this is fast enough to be in the acceptance window Alice accepts the identity of Bob (who is instead the attacker). Notice that this is not a replay: Alice has never received timestamp $t_A$ before. She has generated but never received it, in fact.
This attack shows that the **symmetry** of the key is **dangerous** if there is **no information in the ciphertext** about who are the intender **sender** and **receiver**. As a matter of fact, it is enough to specify A or B, as far as Alice and Bob agree on what they expect to see in the message (even one bit would suffice, to indicate the first in alphabetical order, for example).

### 4.1.3 Key exchange

**Authentication** is always **preliminary** to some other task that requires **identification**. However, it is useless to adopt a strong-authentication protocol and then start an unprotected remote session:

$$A \to B : E_K(ts_A, B)$$
$$A \to B : M_A$$

The attacker can intercept message $M_A$ and substitute it with a different one. If used in this way, strong-authentication becomes the same as OTPs: the attacker, if in the middle, can impersonate the claimant once.
This can be easily solved by **exchanging** a **new (session) key** while **identifying**. Since strong authentication is based on encrypted responses, one simple technique is to **enclose** the **new key inside** the **ciphertext**. Notice that this is **not possible** with **password-based authentication**, unless we use passwords to derive cryptographic keys. We obtain the following.

**ISO/IEC 11770-2 protocols** **One-pass unilateral key-establishment**

$$A \to B : E_K(ts_A, B, k_s)$$

,where $ts_A$ is a timestamp or a sequence number (then B will check is $ts_B - ts_A < W$).
**Two-pass unilateral key-establishment**

$$B \to A : N_B$$
$$A \to B : E_K(N_B, B, k_s)$$

**Two-pass mutual key-establishment**, here Alice and Bob authenticate each other.

$$A \to B : E_K(ts_A, B, k_s^A)$$
$$B \to A : E_K(ts_B, A, k_s^B)$$

,where $ts_A$, $ts_B$ are either timestamps or sequence numbers. The session key is then computed as a function (for example a bitwise XOR) of the two subkeys $k_s = f(k_s^A, k_s^B)$.
**Three-pass mutual key-establishment**

$$\begin{aligned} &B \to A : N_B \\ &A \to B : E_K(N_A, N_B, B, k_s^A) \\ &B \to A : E_K(N_B, N_A, k_s^B) \end{aligned}$$

Notice that we have the same protocol, but now we're adding the session keys. As above, the session key is computed as $k_s = f(k_s^A, k_s^B)$.

### 4.1.4 Server-based protocols

The point-to-point protocols we have studied so far assume a **pre-shared key** $K$ between Alice and Bob. This **does not scale** if we have many users as we should share different keys for any possible pairs (meaning a squared number of keys with respect to the number of users, and specifically $n(n-1)/2$). Moreover, it is **unclear how** a **new user** might **establish** a **new key** for each different existing user.
Determining a shared-key for symmetric key cryptography, and securely obtaining the public key for public key cryptography, can be solved using **trusted intermediary**. For symmetric key cryptography, the trusted intermediary is called a **Key Distribution Center** (KDC), while for public key cryptography, it is called **Certification Authority** (CA).
The **Key Distribution Center** (KDC) is a service for distributing new session keys to any pair of user asking for them. The KDC shares one key with each user U, noted $K_U$, and users do not directly share keys among them. When a new user Alice is added she just needs to register to the KDC and get her key $K_A$.

**Communication between A and B** Suppose that A and B want to communicate using symmetric key cryptography and a trusted KDC. They only know their individual keys, $K_{A-KDC}$ and $K_{B-KDC}$, respectively, for communicating securely with the KDC. One of the most famous key-establishment protocol based on symmetric-key cryptography and on a KDC is the Needham-Schroeder shared-key protocol:

$$\begin{aligned} &1. \quad A \to KDC &&: \quad A, B, N_A \\ &2. \quad KDC \to A &&: \quad E_{K_A}(k_s, B, N_A, E_{K_B}(k_s, A)) \\ &3. \quad A \to B &&: \quad E_{K_B}(k_s, A) \\ &4. \quad B \to A &&: \quad E_{k_s}(N_B) \\ &5. \quad A \to B &&: \quad E_{k_s}(N_B - 1) \end{aligned}$$

We describe the protocol in detail:

1. Alice sends a request to KDC for communicating with B. The request contains a nonce $N_A$;
2. KDC sends a message encrypted under Alice's key containing a new session key $k_s$, the name of Bob and the nonce (which are both checked by Alice), plus a message encrypted for Bob;
3. Alice forwards the message encrypted for Bob to Bob;
4. Bob decrypts the message and obtains the pair $(k_s, A)$ representing a new session key $k_s$ to be used with user A. He sends a nonce $N_B$ encrypted under the new session key to check that Alice knows the key (this is called the key confirmation step);

5. Alice decrypts the nonce sent by Bob, decrements it by 1 and sends it back encrypted. In this way she proves the knowledge of the session key.

To understand what security guarantees are provided by the protocol we reason on the various entities involved:

- KDC: it generates two messages encrypted under Alice and Bob keys. In this way the KDC is guaranteed that the new session key will only be accessed by Alice and Bob and each of them will know who is the expected counterpart, since the relative identifiers are encrypted together with the session key. For example, when Alice decrypts her message she knows that the intended party is Bob, since B is included in the encryption;
- Alice: she challenges the KDC with a nonce to avoid an attacker in the middle can replay old messages from the KDC. In this way she is guaranteed that the key is a new one and that the KDC is in fact answering her request. The first two messages are similar to the two-pass unilateral key-establishment we already discussed;
- Bob: he receives a message (from Alice) encrypted by the KDC. This message does not contain any time-variant parameter so it might be a replay of an old message. To mitigate this, Bob challenges Alice to prove she knows the session key by sending a nonce encrypted under the key. Only knowing $k_s$ it is possible to decrypt the nonce and send back its value, decremented by 1.

This approach makes sense in a **local**, controlled **environment** (such as computer science department or a private company), but it **does not scale** to a **wide area network** such as the Internet, where the entities cannot be all registered under a centralized server.

## 4.2 Diffie-Hellman key agreement protocol

We have seen that sharing secret keys is fundamental for authenticating and running secure sessions. The **Diffie-Hellman key agreement protocol** allows for 'magically' **establishing** a **fresh secret key** between Alice and Bob with **no need of pre-shared keys** or **secrets** (using an insecure channel). The key can then be used in a symmetric key cipher.
The scheme is based on **discrete logarithms**. We choose a prime number $p$ and a generator $a$ of $\{1, \ldots, p-1\}$. A generator $a$ is a number such that $\{a^1 \mod p, \ldots a^{p-1} \mod p\} = \{1, \ldots, p-1\}$. In other words, if we rise $a$ to all the powers $1, \ldots, p-1 \mod n$, we obtain all such numbers. In this sense, $a$ can be used to generate the group.

**Example.** *If* $a = 5$ *and* $p = 23$*, we have that* $5^1 \mod 23 = 5$, $5^2 \mod 23 = 2$, .., $5^{22} \mod 23 = 1$

When this happens, we can define the discrete logarithm modulo $p$ of any number $1 \leq b \leq p-1$ as follows: $log_a b$ is the power $i$ such that $a^i \mod p = b$. It is possible to prove that for any prime $p$ there always exists at least one generator $a$ [1, fact2.132].

**Example.** *If* $b = 2$*, then* $5^2 \mod 23 = 2$.

**Computing** the discrete logarithm modulo $p$ is **infeasible** for a big prime $p$ such that $p-1$ has at least a big prime factor (this is to prevent the use of the Pohlig–Hellman algorithm which works well only when prime factors of $p-1$ are small). Diffie-Hellman protocol for key agreement picks one of such big primes $p$ and a generator $a$ of $\{1, \ldots, p-1\}$. The prime $p$ and the generator $a$ are public. Alice and Bob generate two secrets $S_A$, $S_B$ and run the following protocol:

$$A \to B : a^{S_A} \mod p$$
$$B \to A : a^{S_B} \mod p$$

Alice and Bob compute the new key respectively as $(a^{S_B})^{S_A} \mod p$ and $(a^{S_A})^{S_B} \mod p$, that clearly give the same value. Computing the secrets $S_A$, $S_B$ from the exchanged messages amounts to compute the discrete logarithm, that we have assumed to be infeasible. Thus, an attacker eavesdropping the exchanged traffic will never be able to compute the new exchanged key.

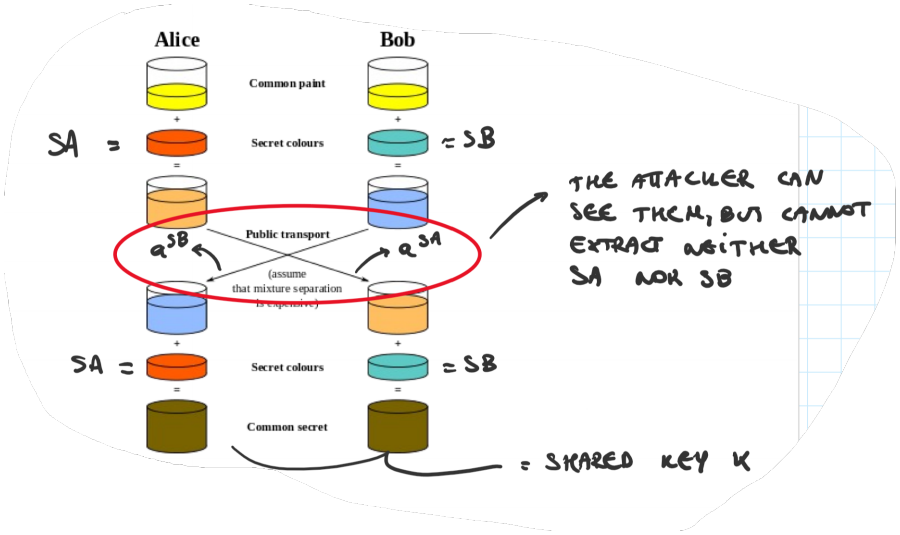



Figure 64: Diffie-Hellman protocol: scheme

**Example.** *The prime $p = 23$ and the generator $a = 5$ are public. Alice chooses $S_A = 6$, and Bob chooses $S_B = 15$. What is the new secret key?*
*We have*

$$A \to B : a^{S_A} \mod p = 5^6 \mod 23 = 8$$

*and*

$$B \to A : a^{S_B} \mod p = 5^{15} \mod 23 = 19$$

*Now, Alice and Bob compute the new key respectively as*

$$(a^{S_B})^{S_A} \mod p = 19^6 \mod 23 = 2$$

*and*

$$(a^{S_A})^{S_B} \mod p = 8^{15} \mod 23 = 2$$

*Alice and Bob now share the secret key $k = 2$.*

### 4.2.1 Man-in-the-middle

Even if Diffie-Hellman protocol has the nice above property about **key confidentiality**, the fact it is not based on any pre-shared secret makes it completely **vulnerable** to **active attackers** that are able to **intercept** and **introduce messages** on the network. In particular, an attacker can mount a **man-in-the-middle attack** where he is able to **impersonate** Alice with Bob and Bob with Alice. This is easily achieved by establishing two different keys with them, so that he can be in the middle in the subsequent session. The attack works as follows:

$$\begin{array}{lllll} A & \rightarrow & E(B) & & : a^{S_A} \mod p \\ A & \leftarrow & E(B) & & : a^{S_E} \mod p \\ & & E(A) & \rightarrow \quad B & : a^{S_E} \mod p \\ & & E(A) & \leftarrow \quad B & : a^{S_B} \mod p \end{array}$$

Now Alice and Bob respectively share with the attacker keys $k_s^1 = a^{S_A S_E} \mod p$ and $k_s^2 = a^{S_B S_E} \mod p$. Next messages, encrypted under such keys, will all 'pass through' the attacker as follows:

$$\begin{array}{lllll} A & \rightarrow & E(B) & & : E_{k_s^1}(M_A) \\ & & E(A) & \rightarrow \quad B & : E_{k_s^2}(M_A) \\ & & E(A) & \leftarrow \quad B & : E_{k_s^2}(M_B) \\ A & \leftarrow & E(B) & & : E_{k_s^1}(M_B) \end{array}$$

, where we recall that $E(B)$ represents the attacker who pretends to be B.
Alice and Bob believe to communicate in a secure, encrypted way, but the attacker is in fact decrypting and re-encrypting any message they exchange.

### 4.2.2 Summary

**Diffie-Hellman protocol** is **secure** only **against passive attackers**. The absence of any pre-shared secret makes it impossible to authenticate parties. An attacker can easily pretend to be one party and mount man-in-the-middle attacks, as shown above. This protocol can be made secure using digital signatures (so to provide authentication).

## 4.3 Strong authentication based on asymmetric-key cryptography

We have seen that one technique to authenticate and perform session key establishment is to have a **centralized KDC service** that shares a symmetric key with any registered users. While this solution makes sense in a **local**, controlled **environment** (such as a computer science department or a private company), it **cannot scale** to a wide area network such as the Internet, where entities cannot be all registered under a centralized server.
**Asymmetric-key protocols** are more **suitable** in this setting, as they **allow** for **authentication** and **key-establishment without** the **presence** of **on-line servers**. We discuss how the previous techniques need to be modified when asymmetric-key cryptography is employed.

### 4.3.1 Asymmetric key authentication protocols

Let us start from a basic, flawed, unilateral authentication protocol based on timestamps inspired to the one we proposed for symmetric-key encryption:

$$A \rightarrow B : E_{PK_B}(A, t_A)$$

Alice sends her name and a valid timestamp to Bob, both encrypted under Bob's public key $PK_B$. This protocol is far from being correct, since any user can send the very same message to Bob, given that $PK_B$ is public. So, for example, the attacker can easily pretend to be Alice as follows:

$$E \rightarrow B : E_{PK_B}(A, t_E)$$

Asymmetric-key encryption never proves the knowledge of a secret as it only requires the knowledge of $PK_B$. Decryption, instead, can be done only when the private key is known. Thus, a correct unilateral authentication protocol can be obtained by challenging Alice to decrypt something.

The more natural way to achieve this is to adopt a Nonce-based authentication scheme:

$$B \to A : E_{PK_A}(N_B, B)$$
$$A \to B : N_B$$

Bob sends a nonce $N_B$ encrypted together with his identifier under the public key of Alice. Only knowing Alice's private key, it is possible to decrypt the nonce and send back its value in the clear. This proves the knowledge of a secret that only Alice is assumed to possess and provides authentication. The presence of the time-variant nonce prevents possible replays. The reason why it is important to specify the identifier in the first message is to prevent man-in-the-middle attacks. We illustrate this subtle issue on the following mutual-authentication protocol.
The unilateral authentication protocol above can be extended using the Needham-Schroeder public-key protocol (omitted in these notes, for details refere to these notes[3]) in order to provide mutual authentication.

### 4.3.2 Signature-based authentication protocols

Another way to provide **authentication** and **key-establishment** using **asymmetric-key** is combining asymmetric-key **encryption** and **digital signatures**. Encryption is needed to protect key confidentiality while signature gives authentication.
We start from the the basic, flawed unilateral protocol:

$$A \to B : E_{PK_B}(A, t_A, k_s)$$

However, in this case everybody knows the $PK_B$, so we add a signature from Alice to prove Alice identity: this, in fact, replaces Alice identifier. The timestamp can be sent in the clear. We include Bob's identifier in the signature since, as we have discussed, it is important to specify the intended receiver of the authenticated exchange. We obtain the following protocol:

$$A \to B : t_A, E_{PK_B}(k_s, sign_A(B, t_A, k_s))$$

The problem with this solution is that the message to encrypt will be typically bigger than the size of one block (for RSA, bigger than the modulus since the signature is already, by itself, as big as the modulus). Implementing this protocol would amount to adopt some encryption mode such as CBC, with strong integrity guarantees.
A **solution** might be to **separate encryption** and **signature** as follows:

$$A \to B : t_A, E_{PK_B}(k_s), sign_A(B, t_A, k_s)$$

A **symmetric key** is typically much **smaller** than **one encryption block** for asymmetric key cryptography (for example, we might have a 1024 bits RSA modulus and a 128 or 256 bits AES symmetric key). The problem with this solution is that signatures sometimes allow for computing the signed message. For example, a "raw" signature implemented as RSA encryption under the private key (with no hash) can be decrypted using the public key, giving the message in the clear and thus the value of the session key. This protocol can be adopted only when the signature scheme prevents the computation of the signed message.
A general solution is thus to **first encrypt** and then **sign**.

$$A \to B : t_A, E_{PK_B}(k_s, A), sign_A(B, t_A, E_{PK_B}(k_s, A))$$

Since signature can be implemented using hashes we do **not have length issues** with this protocol. Notice that Alice identifier has been included in the encrypted message since, as usual,

[3] https://secgroup.dais.unive.it/teaching/cryptography/asym-key-authentication/

we want messages to be explicit about the parties involved in the protocol. More specifically, this **prevents** that an **attacker intercepts** the message and signs the (encrypted) key himself. This could be exploited in settings where the protocol gives credit (such as recharging a prepaid card, or getting an award for submitting the solution to a problem). By signing the session key the attacker will get credit for whatever is sent by Alice encrypted under $k_s$. **Adding** the **identifier** clearly **prevents** this **attack** since Bob will check that the identifier of the signature is the same as the one included in the encrypted message.

**Remark** We have seen in numerous examples that when developing an authentication protocols it is important to specify the identifier which is not already implicit in the key. For example if we encrypt under Bob's public key it is good to specify A. If Alice is signing it is good to include B in the signature.

**X.509 strong authentication protocol** If we run the above unilateral authentication protocol twice we basically obtain the X.509 **strong two-way authentication protocol**:

$$A \to B : t_A, E_{PK_B}(k_s^A, A), sign_A(B, t_A, E_{PK_B}(k_s^A, A))$$
$$B \to A : t_B, E_{PK_A}(k_s^B, B), sign_B(A, t_B, E_{PK_A}(k_s^B, B))$$

Alice and Bob compute a session key as $k_s = f(k_s^A, k_s^B)$.
**Mutual authentication** can also be achieved based on nonces as follows:

$$B \to A : N_B$$
$$A \to B : N_A, E_{PK_B}(k_s^A, A), sign_A(B, N_A, N_B, E_{PK_B}(k_s^A, A))$$
$$B \to A : E_{PK_A}(k_s^B, B), sign_B(A, N_B, N_A, E_{PK_A}(k_s^B, B))$$

,which is very close to the X.509 strong three-way authentication protocol described here[4].

## 4.4 Key management

We now discuss how keys are managed, stored and checked by parties and KDC servers.

### 4.4.1 Symmetric key management

We have seen that practical authentication protocols based on symmetric keys require the presence of a centralised trusted party that shares one long-term key with each possible user. It is important to have a **way** to securely **deal** with these **keys** so that an attack to the KDC system would not necessarily compromise the keys of all users.
An interesting solution to this problem is provided by **symmetric key certificates**. The **trusted party** possesses a **master key** $K_M$ that it only knows. When a new user $U$ is registered, the respective **long-term key** $k_U$ is **generated** and is **encrypted** under the **master key** together with the **identifier** and additional information such as the key lifetime, type, etc. For example, $SCert_U = E_{K_M}(U, k_U, L)$ certifies that user $U$ has key $k_U$ with lifetime $L$.
These **certificates** can be distributed to users together with their keys and users can freely distribute their certificates to other users, since the encryption under the master key protects the confidentiality of the enclosed keys. The trusted party can completely forget about the keys and ask for certificates when needed.

[4] https://www.itu.int/rec/T-REC-X.509

### 4.4.2 Asymmetric key management

The idea of certificates is even more important for protocols based on **asymmetric keys**. In this case, in fact, it is crucial that **key management** is completely **decentralised** since we do not want any on-line centralised servers available at each protocol execution.
First notice that the relevant property here is **authenticity** of **public keys**. If Alice and Bob want to communicate they need a way to check that public key is the one truly associated with the opposite party. Suppose Alice wants to send Bob a secret message M. It is insecure for Bob to simply send his public key to Alice:

$$\begin{aligned} B &\to A : PK_B \\ A &\to B : E_{PK_B}(M) \end{aligned}$$

In fact, an attacker can replace $PK_B$ with his own key:

$$\begin{aligned} E(B) &\to A &: PK_E \\ A &\to E(B) &: E_{PK_E}(M) \end{aligned}$$

The attacker can then decrypt the secret message encrypted under his public key.
We need a **mechanism** that allows Alice to **check** that the **received key** is the one belonging to Bob. Public key certificates contains the same informations as the symmetric key ones but, instead of being encrypted under a symmetric master key, they are signed by a Certification Authority (CA), a trusted entity that certifies the authenticity of user's public keys. For example, $Cert_B = B,\ PK_B,\ L,\ sign_{CA}(B, PK_B, L)$.
If Alice knows the public key of the CA and Bob sends this certificate, then Alice can check the validity of the key and its association with Bob (B). The protocol becomes:

$$\begin{aligned} B &\to A : Cert_B \\ A &\to B : E_{PK_B}(M) \end{aligned}$$

The attacker cannot change the public key as he is not able to forge a signature from the CA.

**Certificate chains** Having just **root CA's** signing any certificate in the world **does not scale** well. It is useful in practice to have different **levels** of **certifications** (maybe associated with countries, states, cities, etc.) so that the end user can go the 'local' CA. To deal with this **hierarchical organisation of CA's** we need to be able to check certificate chains: the root CA certifies the next CA in the hierarchy and so on all the way to the end user. Once we provide the whole **chain**, it is enough to **trust** the **root public key** in order to check all the certificates in the path to the end user.

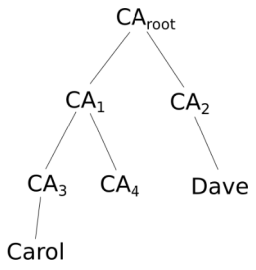



Figure 65: Certificate chain: example

Above we illustrate two end users Carol and Dave certified by two different CAs under the same root CA. If Carol wants to communicate with Dave, she needs to send Dave the certificate chain

from the root CA to herself: $CA_1$ certificate signed by $CA_{root}$, $CA_3$ certificate signed by $CA_1$ and Carol's certificate signed by $CA_3$.

**Further extensions** The basic **hierarchal tree organisation** illustrated above can be extended to multi-trees, where many root CAs are present (as typically happens in browsers): if any users know many root CAs it is enough a chain from one of the roots to a leaf to certify the corresponding user. To avoid storing all the root certificates, root CAs might certify each other so that any root CA is 'reachable' by any other.
Interestingly, there are solutions where no centralised authorities are required. In **Pretty Good Privacy** (PGP) it is proposed a **web-of-trust**: any **user** can **trust** other **users**. The level of trust might be such that if Alice trusts Bob, any other user certified by Bob is trusted by Alice. In a sense, each user can 'elect' other users as CA's that can be trusted when signing certificates for other users. This can be depicted as a **general graph** with **no roots**: a **path** in the graph represents a **chain of certificates** from oner user to the other. If we trust the starting user, we can successfully verify the identity and public key of the final user in the path.

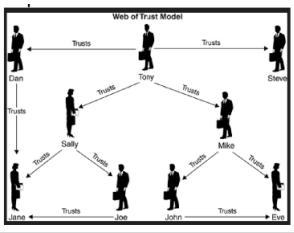



Figure 66: PGP: web of trust