Ca' Foscari University

Faculty of Computer Science

Cryptography

Afternotes

Author

Francesco Vivian

A.Y. 2020-2021

Lecture 1

Definition 1. Security: it generically refers to the possibility of "protecting" information, which is either stored in a computer system or transmitted on a network.

Whatever will be shown works in both situations.
To decide whether a computer system is "secure", you must first decide what secure means to you, then identify the threats you care about. Some threats are: cyberterrorism, denial of service, modified databases, virus, identity theft, stolen customer data, equipment theft, espionage.

There are different aspects to protect through security properties:

Property 1. Authenticity: an entity should be correctly identified.

Example 1. Some examples of authenticity:

- login process for authenticating a user (User Identification)
- a digital signature allows for authenticating the entity originating a message (Message Authentication)

Property 2. Confidentiality (secrecy): information should only be accessed (read) by authorized entities.

- Confidential information is not disclosed to unauthorized individuals (Data confidentiality)
- Individuals control what information related to them may be collected and stored and by whom that information may be disclosed (Privacy)

Example 2. Some examples of confidentiality:

- the person that accesses a database should be authorized to access the data
- personal privacy, my private data should be protected while browsing the web

The "access control" for confidentiality:

- use the "need to know" basis for data access. How do we know who needs what data? (Approach: access control specifies who can access what). How do we know a user is the person she claims to be? We need her identity and we need to verify this identity (Approach: identification and authentication).

- Analogously, the "need to access/use" is the basis for access to physical assets (access to a computer room, use of a desktop)

Confidentiality is difficult to ensure and easy to assess in terms of success: it is binary in nature (Yes/No).

Property 3. Integrity: information should only be modified by authorized entities.

- information and programs are changed only in a specified and authorized manner (Data integrity)
- a system performs its intended function, free from unauthorized manipulation (System integrity)

Example 3. We should not alter bank accounts and IoT device firmware.

If we don't have integrity, we also don't have confidentiality (with integrity I want only authorized users to be able to modify information, with confidentiality I want only authorized users to be able to see information, modifying includes seeing). Integrity is more difficult to to measure than confidentiality, it is non binary (it has degrees of integrity) and it is content-dependent (it means different things in different context)

Example 4. A quotation from a politician, we can preserve the quotation (data integrity) but mis-attribute (origin integrity), like *Y said that* instead of *X said that.*

Property 4. Availability: information should be available/usable fastly by authorized users.

Example 5. It is important to guarantee reliability and safety. Apart from attacks, availability might be loss in case of faults (we need to use fault-tolerant techniques). We need availability in case of a remote surgery for example (good QoS).

We can say that an asset (a resource) is available if:

- it provides a timely request response
- it provides fair allocation of resources (no starvation)
- it is fault tolerant (no total breakdown)
- it is easy to use in the intended way
- it provides controlled concurrency (concurrency control, deadlock control, ..)

Property 5. Non-repudiation: an entity should not be able to deny an event.

Example 6. Having sent/received a message. This property is crucial for e-commerce, where "contracts" should not be denied by parties.

Other properties that are not addressed in detail are: fairness of contract signing, privacy, anonimity and unlinkability, accountability, ..

Typical attacks

We will now see some typical attacks. We assume that information is flowing from a source to a destination (e.g.: reading data is a flow from the data container to a user, writing is a flow from a user to the file system).

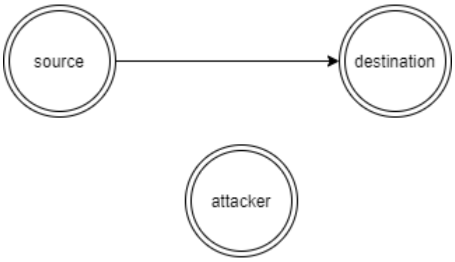


Figure 1: Expected information flow

Malicious users might try to subvert the properties previously mentioned in many different ways. We will now give a general classification depending on how an attacker might interfere on the expected flow of information (Figure 1).

Definition 2. Interruption: the attacker stops the flow of information (Figure 2). The attacker interrupts a service, it breaks system integrity and availability.

Some examples of interruption:

Example 7.

- the destruction of a part of the hardware
- canceling of programs or data files
- the destruction of a network link
- a denial of service (DoS) that makes the system/network unusable

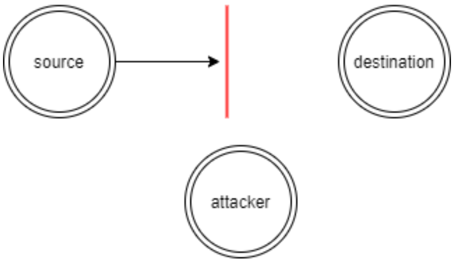


Figure 2: The attacker interrupts the flow of information

Definition 3. Eavesdropping (interception): the attacker gets unauthorized access to the information (depicted as an additional flow towards the attacker).

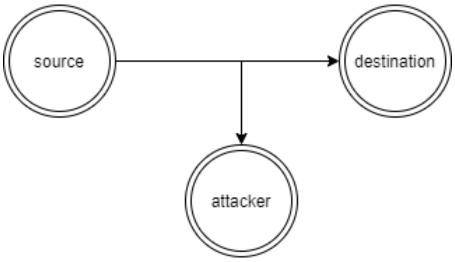


Figure 3: The attacker intercepts information

Interception is an attack to confidentiality, these attacks are hard to detect.

Example 8.

- unauthorized copies of files or programs;
- interception of data flowing in the network (a credit card number).

Interception attacks are hard to detect because source and destination don't notice any change (differently from interruption, where destination don't receive the flow).

Definition 4. Modification: the attacker intercepts the information and make unauthorized modification.

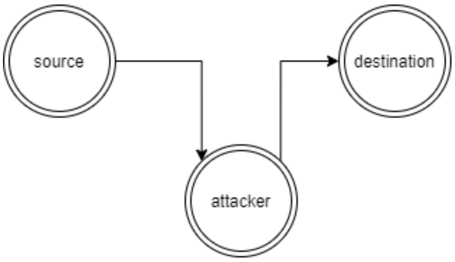


Figure 4: The attacker modifies information

In this case destination might notice that something is wrong.

Example 9.

- unauthorized change of values (e.g.: of a database);
- unauthorized change of a program;
- unauthorized change of data flowing in a network;
- `A` redirects `S`'s bank transfer to herself (either in the browser or in the network, man in the middle).

Definition 5. Forging (falsification): the attacker inserts new information in the system (usually related to impersonation since the attacker lets the destination believe the information is coming from the honest source).

Forging is an attack to *authenticity, accountability* and *integrity*.

Example 10.

- addition of messages in the network;
- addition of a record in the database.

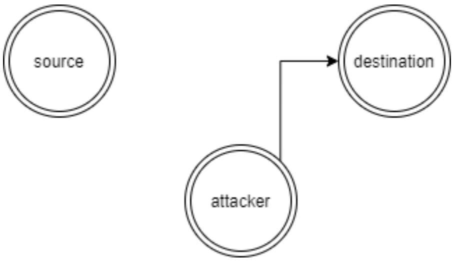


Figure 5: The attacker forges new information

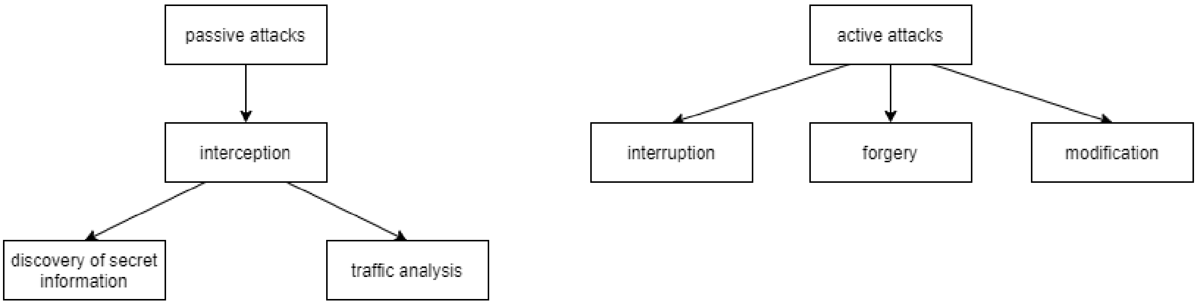


Figure 6: Classification of different types of attacks

Example 11. Suppose a bank B is using the following simple protocol to allow a bank transfer from user Alice (A):

`A` → `B: sign_A(`"please pay Bob 1000€"`)`

`sign_A` is some "signature" mechanism to ensure that the message really comes from Alice (thus the attacker cannot generate valid signed messages from Alice).

Let's suppose Bob is the attacker, he intercepts the whole message and repeats it as many times as he want, without modifying it. This attack (called replay) consists of an interception plus forging (in this case the message is just re-sent as it is). Bob obtains many bank transfers by just re-sending message M.

Example 12. Program modification: It is an attack to confidentiality, Bob modifies a program that is used by Alice, such a program apparently works normally, however it changes (e.g. the access rules of the users that are executing it, in this case it is called *Trojan horse*). Bob waits that Alice uses the program and copies all the files of Alice in his home directory.

Cryptography

The term *cryptography* comes from the greek and means "hidden writing". It is a way to protect the information when the environment is insecure. For example it is used when the information is sent on a network such as internet or when the system does not support sufficient protection mechanisms.

Definition 6. Encryption: a *plaintext* (message) is transformed using some rules (encryption algorithm) in a ciphertext.

Definition 7. Decryption: the plaintext is reconstructed starting from a ciphertext.

The decryption has to be simple for the receiver and unfeasible for an attacker. The information is encrypted in the source and travels to the destination where it will be decrypted. In order to do this there are two possible solutions:

1. only the sender (Alice) and the receiver (Bob) know the encryption algorithm. If the attacker, by looking at the flow of information, is able to guess which algorithm is used, then he will be able to decrypt all the messages sent;

2. the encryption algorithm is public and Alice and Bob share some information (the encryption key) non accessible by the attacker. If the attacker doesn't have the encryption key, it is unfeasible to decrypt the messages.

The second solution is better because it is simpler to distribute only one key and if there is an attack it is easier to change key instead of a whole algorithm.

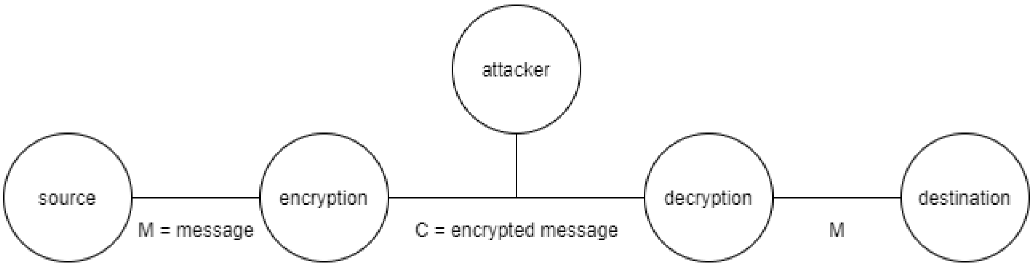


Figure 7: Encryption with a shared key

We want to build a secure channel to be able to exchange the encryption key.

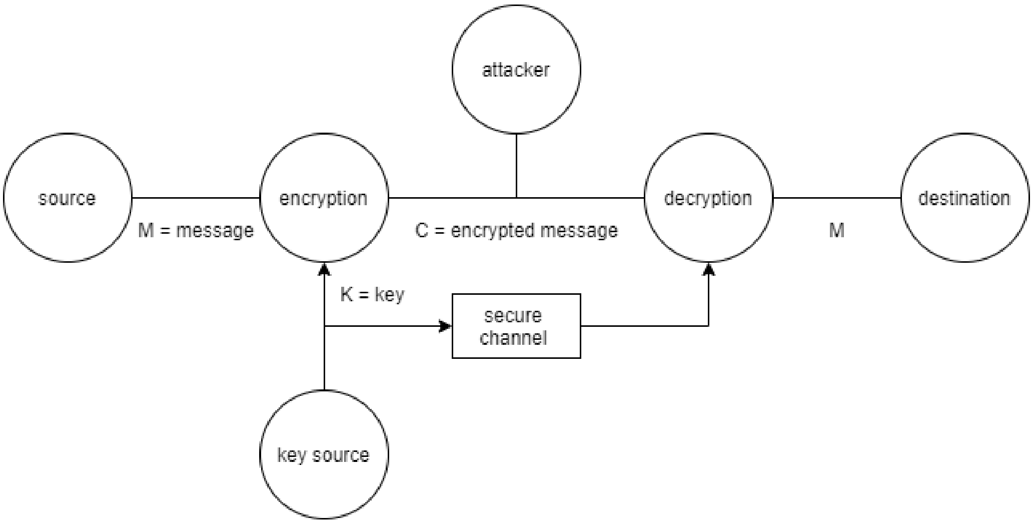


Figure 8: Encryption with a shared key and the secure channel

This is called *symmetric key cipher* (symmetric because source and destination use the same key).

One of the first encryption algorithms was used by Julius Caesar. In the Caesar Cipher all the letters are permuted using a certain rule (each letter is substituted by the one 3 positions ahead in the alphabet)

$$\begin{aligned} \mathtt{A} &\rightarrow \mathtt{D} \\ \mathtt{B} &\rightarrow \mathtt{E} \\ \mathtt{C} &\rightarrow \mathtt{F} \\ &\dots \\ \mathtt{Z} &\rightarrow \mathtt{C} \end{aligned}$$

In this case the algorithm is the Caesar Cipher and the key is 3.

Lecture 2

The idea behind this course is to build step-by-step a system that is stronger and stronger until we'will arrive to systems that are used in practice.

Defining a cipher means to define an encryption algorithm and a decryption algorithm. A **cryptosystem** (or **cipher**) can be defined as a quintuple `(P,C,K,E,D)` where:

- `P` is the set of plaintexts (i.e. all the Italian words);
- `C` is the set of ciphertexts;
- `K` is the set of keys (we can have more than one key);
- `E:K×P→C` is the encryption function;
- `D:K×C→P` is the decryption function.

Let `x∈P, y∈C, k∈K`, we will write E_k(x) and D_k(y) to denote E(k,x) and D(k,y), the encryption and the decryption under the key k of x and y respectively.

We require two properties for each cipher that uses a shared key:

Property 6. $D_k(E_k(x)) = x$, decrypting a ciphertext with the right key gives the original plaintext.

Property 7. computing k or x given a ciphertext is infeasible, so complex that cannot be done in a reasonable time.

All the ciphers we will discuss have the first property, only "secure" ciphers have the second one (Caesar cipher doesn't). If someone, one day, will prove that P=NP, then most of the ciphers won't be secure anymore.

Example 13. Referring to the Caesar cipher, we can define the encryption function as "the letter three positions ahead (of our letter x) in the alphabet" or `(x+3) mod26` and the decryption function as "the letter three position behind (of the letter to be decrypted)" or `(x-3) mod26`. Our key k=3.

If we find the message `"BHV BRX PDGH LW"` and we know that it is encrypted with the Caesar cipher we can easily decrypt it into `"YES YOU MADE IT"` just going back 3 positions. We can notice that we have two "B" and they correspond to the same decrypted letter "Y", in monoalphabetic ciphers this is a strong weakness.

Proof of the first property applied to the Caesar cipher. We have to prove that:

$$\begin{aligned} x &= D_k(E_k(x)) && (1)\\ &= ((x+3) \text{ mod}26 - 3) \text{ mod}26 && (2)\\ &= ((x+3) - 3) \text{ mod}26 && (3)\\ &= x \text{ mod}26 && (4)\\ &= x && (5) \end{aligned}$$

Since we have the modules repeated twice we can keep only the external one. ■

To prove this property we can apply this reasoning to every cipher.

Definition 8. Kerckhoffs' principle: a cipher should remain secure even if the algorithm becomes public.

Kerckhoffs rules (1883):

- The system should be, if not theoretically unbreakable, unbreakable in practice;
- The design of a system should not require secrecy, and compromise of the system should not inconvenience the correspondents (Kerckhoffs' principle);
- the key should be memorable without notes and should be easily changeable;
- the cryptograms should be transmittable by telegraph;
- the apparatus or documents should be portable and operable by a single person;
- the system should be easy, neither requiring knowledge of a long list of rules nor involving mental strain.

The Caesar cipher is clearly insecure (it will be proved later) since once the cipher has been broken any previous exchanged message is also broken (as the cipher works the same way), the key should be changed and it is assumed to be the only secret.

Shift cipher

We can extend the Caesar cipher to a shift cipher with a generic key k, we can choose any key in the range $0 \leq k \leq 25$. For simplicity we will consider letters as numbers (`A=0, B=1, .., Z=25`), this means that $P = C = K = Z_{26}$ (Z_{26} is for all the integers between 0 and 25). For the encryption and decryption function we have:

$$\text{E}_k(x) = (x+k)\text{mod}26$$
$$\text{D}_k(y) = (y-k)\text{mod}26$$

The Caesar cipher is a subcase of a shift cipher with k=3. In a shift cipher is useless to have k=0 because we would end up with equals plaintexts and ciphertexts.

Example 14. Considering k=10, it gives the following substitution:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
             ↓
KLMNOPQRSTUVWXYZABCDEFGHIJ
```

Proof of the first property applied to a shift cipher. We have to prove that:

$$\begin{aligned} x &= D_k(E_k(x)) && (1)\\ &= D_k((x+k) \bmod 26) && (2)\\ &= ((x+k) \bmod 26 - k) \bmod 26 && (3)\\ &= ((x+k)-k) \bmod 26 && (4)\\ &= x \bmod 26 && (5)\\ &= x && (6) \end{aligned}$$

■

Note that Z_{26} is a group under the addition (but not under the multiplication).

Definition 9. A **group** $\langle G, * \rangle$ is a set G together with a (closed) binary operation * on G such that:

- the operator is associative ($(x * y) * z = x * (y * z)$ for all x,y,z in $\langle G, * \rangle$);
- there is an element e∈G such that $a * e = e * a = a$ for all a∈G. Such an element is the identity element;
- for every a∈G, there is an element b∈G such that $a * b = e$. This b is said to be the inverse of a with respect to *. The inverse of a is sometimes denoted as a^{-1}.

The set $\langle Z, + \rangle$, which is the set of integers under addition, forms a group:

- addition is associative ($(x + y) + z = x + (y + z)$ for all x,y,z in $\langle Z, + \rangle$);
- the identity element is 0, since $0 + a = a + 0 = a$ for any a∈Z;
- the inverse of any a∈Z is -a.

A group that is commutative with an additive operator is said to be an abelian group.

The set $\langle Z, \cdot \rangle$, the set of all integers under multiplication, does not form a group. There is a multiplicative identity 1 but there is no multiplicative inverse for every element in Z.

From now on we will work with abelian groups.

Possible attack to shift ciphers

If I can attack a shift cipher, I can implicitly attack the Caesar cipher.

Example 15. If I am an attacker, I see the message `NGPPS` and I know that it is encrypted using a shift cipher but I don't know which key k is used I can try to get it by trial and error. I start with `k=1` and if decrypting the message I obtain something nonsense, I try with k=2 and so on. with k=4 I will reach that `NGPPS=HELLO`. Is it feasible? Yes because we only have 26 possible to keys to try. This type of attack is called **Brute force**.

In this case the problem with our encryption algorithm is the very small number of keys. Thus the second property doesn't hold (Kerckhoffs' principle: a cipher should remain secure even if the algorithm becomes public). The weakness is that we know that each letter is moved by the same distance.

Substitution cipher

A substitution cipher is a generalization to overcome the previous limitation: instead of moving all the letters by the same distance, now we use a generic permutation of the alphabet to map the letters. For example:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
            ↓
SWNAMLXCVJBUYKPDOQERIFHGZT
```

"`HOME`" becomes "`CPYM`". In this case the key is the complete permutation, otherwise the receiver is not able to decrypt our message. As for the previous ciphers, to decrypt we just apply the inverse substitution.
We have P=C=Z_{26} and K={p|p is a permutation of 0,..,25} with:

- E_k(x)=ρ(x);
- D_k(y)=ρ^{-1}(y).

Can we "Brute force" also this type of ciphers? No, we would have to try 26! keys, which is approximately $4 \times 10^{26} > 2^{88}$. This number of keys is very heavy to brute force, even with powerful parallel computers. We have to find something different to attack substitution ciphers. How can we do it?

It is a monoalphabetic cipher (it maps a letter 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, in Italian, almost all the words end with a vowel, and the vowels (a,e,i,o,u) are easy to identify as they are much more frequent than the other letters.

Let us assume a cryptoanalyst knows the used cipher (e.g. a monoalphabetic substitution cipher) and the language used in plaintext (e.g Italian), but he doesn't know the plaintext and the key.

Example 16. Given a ciphertext C, let us compute the frequency of letters, for example lette S appears 0 times and letter C appears 15 times. Is it possible that the cipher transforms A into S, and Z into C (A is never found and Z is found 15 times)? It is possible but very unlikely.

To compute the statistics for letters in Italian language it has been used a text of 14.998 letters, 1/3 from the book "Pinocchio" and 2/3 from the book "Il nome della rosa". The letters frequency are reported in the following table.

Letter	Absolute frequency	Percentage frequency
A	1714	0.114
B	160	0.011
C	637	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
Total	**14998**	**0.998**

Table 1: Letters statistics for Italian language.

The most used letter is A and the less used is Q. We can also build the graph in figure 9 to represent this statistic.

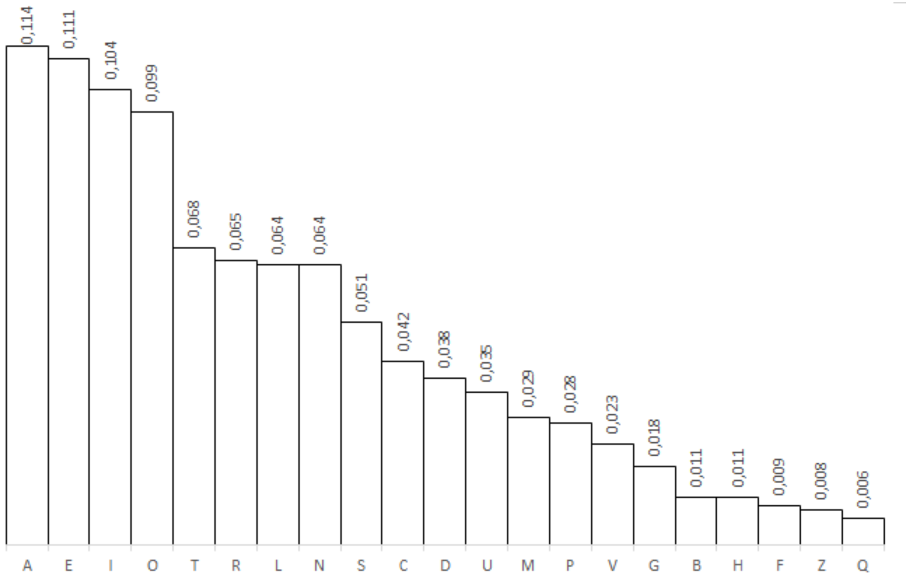


Figure 9: Percentage frequencies of letters in Italian language.

Knowing this, we can study the frequency of each letter in my ciphertext and map them using the frequency of my language.

- In Italian the frequency of letters I and L increases at the beginning of the sentences (articles "il", "lo", "la"), and the frequency of "A", "E", "I", "O" at the end of the words;
- there are words that may appear more frequently (e.g. the word "airplane" in a military message);
- there exist digraph and trigraph that are more frequent.

How can we decrypt a message?

- we order the letters of the ciphertext into decreasing frequencies;
- we substitute with letters in decreasing order as in the corresponding tables (depending on the language);
- there might be mistakes with letters that have the same frequency ("N" and "L" or "H" and "B" in Italian).

We need a long ciphertext in order to obtain reasonable frequencies. It is possible that at the first try we obtain some nonsense words but similar to some words of complete meaning, we can then change letters to correct these words. Repeating some times this step we would end up with the correct plaintext.

We have seen that we can easily break monoalphabetic ciphers applying this method, so the second property doesn't hold. Attacks to substitution ciphers are called **Statistical attacks**

Proof of the first property applied to a substitution cipher. We have to prove that:

$$\begin{aligned} x &= D_k(E_k(x)) && (1)\\ &= D_k(\rho(x)) && (2)\\ &= \rho^{-1}(\rho(x)) && (3)\\ &= x && (4)\\ & && (5) \end{aligned}$$

■

We have seen that monoalphabetic ciphers are prone to statistical attacks, since they preserve the statistical structure of the plaintext. A solution are the polyalphabetic ciphers in which the same symbol is not always mapped to the same encrypted symbol.

Polyalphabetic ciphers

An example of polyalphabetic cipher is the **Vigenére cipher** (XVI century). It works on "blocks" of m letters with a key of length m.

Example 17. The key is `FLUTE` (m=5). The plaintext is split into blocks of length 5 and the key `FLUTE` is repeated as necessary and used to encrypt each block.

```
THISISAVERYSECRETMESSAGE
           +
FLUTEFLUTEFLUTEFLUTEFLUT
           =
YSCLMXLPXVDDYVVJEGXWXLAX
```

Each letter of the ciphertext is given by the sum of the position of the letter in the plaintext plus the position of the letter in the key.

This type of ciphers works better than monoalphabetic ciphers because one letter is not always mapped to the same one unless they are at a distance that is multiple of m.

Formally, P=C=K=Z_{26}^m, where Z_{26}^m is $Z_{26}\times Z_{26}\times..\times Z_{26}$, m times.

- $E_{k1,..,km}(x_1,..,x_m) = (x_1+k_1,..,x_m+k_m)$ mod26;
- $D_{k1,..,km}(y_1,..,y_m) = (x_1-k_1,..,x_m-k_m)$ mod26.

If an attacker knows m, he has to try 26^m possible keys, if m is big enough, it is impossible to brute force it.

Lecture 3

Vigenére cipher cause an almost "flat" distribution of letters frequency, for this reason we would make many mistakes if we try do decrypt a ciphertext in the same way as we do for monoalphabetic ciphers.

As said in the last part of the previous lesson, if we want to brute force this cipher, (assuming we know m) we would have to try 26^m possible keys and that's infeasible, if we don't know m, it would we even worse. It is sufficient to choose m big enough to prevent brute force attacks, note that the number of keys grows exponentially with respect to the length.

Breaking Vigenére cipher

Even if the 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 will see the latter since it is more suitable to be mechanized. Both are based in **recover the length m of the key** and then **recover the key**.

The Friedman method uses statistical measures to recover the length m of the key. We consider the index of coincidence:

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

where f_i is the frequency of the i-th letter in a text of length n, i.e., the number of times it occurs in such text and $p_i = f_i/n$ is the probability of the i-th letter. Intuitively, this measure gives the probability that two letters, chosen at random from the text, are the same. I compute this probability over all the letters.

Example 18. The IC of **"the index of coincidence"** is given by:
c(3*2) + d(2*1) + e(4*3) + f(1*0) + h(1*0) + i(3*2) + n(3*2) + o(2*1) + t(1*0) + x(1*0) = **34**
divided by n(n-1)=21*20 = **420**
which gives us an IC of 34/420 = **0.0809**

The IC of **"bmqvszfpjtcsswgwvjlio"** is given by:
b(1*0) + c(1*0) + f(1*0) + g(1*0) + i(1*0) + j(2*1) + l(1*0) + m(1*0) + o(1*0) + p(1*0) + q(1*0) + s(3*2) + t(1*0) + v(2*1) + w(2*1) + z(1*0) = **12**

divided by n(n-1)=21*20 = **420**
which gives us an IC of 12/420 = **0.0286**

Once I have the IC of my ciphertext, I have to compare it to the ICs of various languages to discover to which it corresponds.

The value of the index is maximum (value 1) for texts composed of just a single letter repeated n times. The value of the index is minimum (value 1/26 ≈ 0.038) for texts composed of letters chosen with uniform probability 1/26.

The index of coincidence is thus a measure of how non uniformly letters are distributed in a text, each natural language has a characteristic index of coincidence, some examples:

English → 0.065
Russian → 0.0529
German → 0.0762
Spanish → 0.0775

We can also use the Friedman method to find moro or polyalphabetic ciphers. We know that if we use frequencies analysis, if frequencies are flat, we have a polyalphabetic cipher, if we have peaks and valleys of frequencies, we have a monoalphabetic cipher. Considering the Friedman method, if the value of the index is minimum ≈ 0.038, we have a polyalphabetic cipher, if it is ≈ 0.065, we have a monoalphabetic cipher (same IC as English, just a permutation of letters).

With the Friedman method we can estimate m, the length of the key in a Vigenére cipher. The idea is to recover m by brute forcing, following this algorithm (in Python):

```
m=1
LIMIT = 0.06 #this is to check that ICs are above 0.06 and thus close to
                                    0.065 (assuming the text is in
                                    English)
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 the length
      found = False
      m += 1
      break
#survived the check, all ICs are above LIMIT
output (m)
```

It works because, once we reach the correct m, all the letters we are considering will be encrypted using the same key, "F" in the below example. So, computing the IC, we will obtain a value similar to the English IC value.

THISISAVERYSECRETMESSAGE
+
FLUTEFLUTEFLUTEFLUTEFLUT
=
YSCLMXLPXVDDYVVJEGXWXLAX

In order to obtain suitable results, we need to have a long enough ciphertext, otherwise we could not be able to succeed.

Now that we have m, we need to find the key. We already said that we cannot brute force it, it would be infeasible. What should we do?

- we divide the text into blocks of length m, as the length of the key (we just found it);
- we need to build new cryptograms with the first letter of each block, one with the second letter and so on;
- we analyse the new cryptograms as before and we find the shift in each position.

We are considering texts composed of letter at distance m from the first one, the second one, and so on. They have different shifts, we need to find the relative right shift.

The idea is to shift one subcipher until the mutual index of coincidence with the first subcipher 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 subciphers and , consequently, between the corresponding letters of the key.

The mutual index of coincidence is defined as:

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

It represents the probability that two letters taken from two texts x and x' are the same.

The following algorithm selects 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 maximum
  key.append(k)     #we append to the list the shift we have found
```

We repeat this for every letter of the key and we obtain the list of relative shifts, for example, if we obtain [0,4,6,3,9], it means that the second letter of the key is equal to the first plus 4, the third is equal to the first plus 6 and so on. But what is the first letter? The final step is to try all the possible 26 letter of the key, that gives us 26 possible keys.

Known-plaintext attacks

Until now we have considered attackers that only know the ciphertext y and try to find either the plaintext x or the key k. It is often the case that an attacker can guess part of the plaintext (e.g., the "standard" header of a message), 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. For example if the attacker is trying to decrypt an email, he can guess the initial par that often starts with "Dear ...". If the attacker knows some plaintexts, this gives him the knowledge of some pairs (x,y) plaintext, ciphertext. Given this, the attacker should be able to decrypt other messages or to recover the key k (no matter which cipher is used).

The **Hill** cipher is a polyalphabetic cipher and it is a generalization of the Vigenére by introducing linear transformations of blocks of plaintext. We have $P=C=Z_m^{26}$, while $K=\{K|K$ is an invertible mod26 matrix $m\times m\}$. The encryption and decryption are the following:

- $E_K(x_1,..,x_m) = (x_1,..,x_m)K$ mod26;
- $D_K(y_1,..,y_m) = (y_1,..,y_m)K^{-1}$ mod26.

Example 19. Let us assume $M=(x_1,x_2)=(5,9)$ and $K = \begin{bmatrix} 5 & 11 \\ 8 & 3 \end{bmatrix}$ (we will only consider 2×2 matrices for simplicity). Thus, $E_K(5,9) = (5,9) \times \begin{bmatrix} 5 & 11 \\ 8 & 3 \end{bmatrix}$ mod26 $= (5 \times 5 + 9 \times 8, 5 \times 11 + 9 \times 3)$ mod26 $= (25 + 72, 55 + 27)$ mod26 $= (97, 82)$ mod 26 $= (19, 4)$

In order to decrypt a message, we need to compute the inverse of the matrix that is our key.

Example 20. We have $(y_1,y_2) = (19,4)$ and $K=\begin{bmatrix} 5 & 11 \\ 8 & 3 \end{bmatrix}$. To compute K^{-1}:

$K^{-1} = \det^{-1}(K) \begin{bmatrix} 3 & -11 \\ -8 & 5 \end{bmatrix}$ mod26 $= \det^{-1}(K)\begin{bmatrix} 3 & 15 \\ 18 & 5 \end{bmatrix}$ mod26.

We assume our matrix K is invertible. In the last step we changed the negative numbers to positive ones considering the mod26.

Now we can calculate $\det(K) = (5 \times 3 - 11 \times 8)$mod26 $= (15 - 88)$mod26 = -73 mod26 = 5. How do we compute $\det^{-1}(K)$? To find the inverse mod26 of 5, we need to find a number in the interval [0,25] that multiplied by 5 mod26 gives 1. The number we are searching is 21, 5×21 mod26 = 105 mod26 = 1. Thus $\det^{-1}(K) = 21$.
Note that it is not always the case that the multiplicative inverse modulo exists, we will discuss this more in detail later on, introducing the public key cryptography and RSA.

Now we can solve $K^{-1} = \det^{-1}(K)\begin{bmatrix} 3 & 15 \\ 18 & 5 \end{bmatrix}$ mod26= $21 \begin{bmatrix} 3 & 15 \\ 18 & 5 \end{bmatrix}$ mod26 $= \begin{bmatrix} 63 & 315 \\ 378 & 105 \end{bmatrix}$ mod 26 $= \begin{bmatrix} 11 & 3 \\ 14 & 1 \end{bmatrix}$. Thus $D_K(19,4) = (19,4)\begin{bmatrix} 11 & 3 \\ 14 & 1 \end{bmatrix} = (19 \times 11 + 4 \times 14, 19 \times 3 + 4 \times 1)$ mod26 $= (265, 61)$ mod26 $= (5, 9)$.

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

Encryption

E$_K$(2,5) = $(2,5)\times\begin{bmatrix} 5 & 11 \\ 8 & 3 \end{bmatrix}$ mod26 = $(2\times5+5\times8, 2\times11+5\times3)$ mod26 = $(10+40, 22+15)$ mod26 = $(50, 37)$ mod26 = $(24, 11)$.

Decryption

K^{-1} =det^{-1}(K) $\begin{bmatrix} 3 & -11 \\ -8 & 5 \end{bmatrix}$ mod26 = det^{-1}(K)$\begin{bmatrix} 3 & 15 \\ 18 & 5 \end{bmatrix}$ mod26.

det(K) = $(5\times3-11\times8)$mod26 = $(15-88)$mod26 = -73 mod26 = 5.

det^{-1}(K) = 21, 5×21 mod26 = 105 mod26 = 1.

K^{-1} = det^{-1}(K)$\begin{bmatrix} 3 & 15 \\ 18 & 5 \end{bmatrix}$ mod26= 21 $\begin{bmatrix} 3 & 15 \\ 18 & 5 \end{bmatrix}$ mod26 = $\begin{bmatrix} 63 & 315 \\ 378 & 105 \end{bmatrix}$ mod 26 = $\begin{bmatrix} 11 & 3 \\ 14 & 1 \end{bmatrix}$.

D$_K$(24,11) = (24,11)$\begin{bmatrix} 11 & 3 \\ 14 & 1 \end{bmatrix}$ = $(24\times11+11\times14, 24\times3+11\times1)$ mod26 = $(418, 83)$ mod26 = $(2, 5)$.

Lecture 4

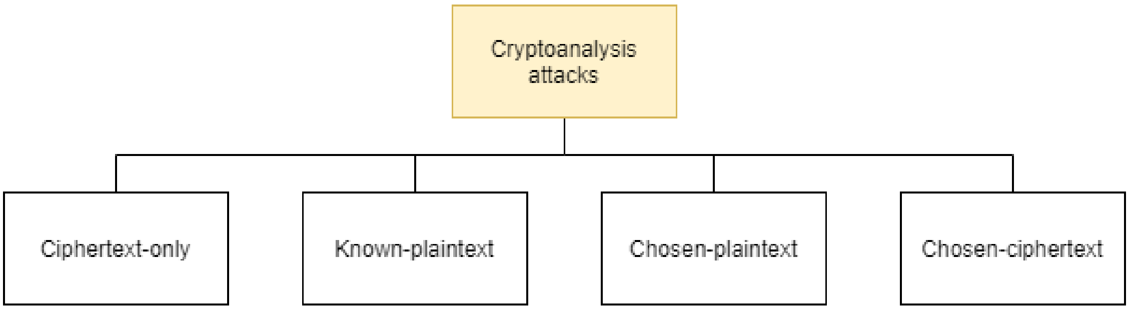


Figure 10: Types of Cryptonalysis attacks.

Up to now we have seen **ciphertext-only** attacks and **known-plaintext** attacks (in the last part of the previous lecture).
In a ciphertext-only attack, the attacker is assumed to have access only to a set of ciphertexts (e.g., monoalphabetic ciphers, polyalphabetic ciphers), the limit is that it is easy to find the correspondence between letters in the plaintext and in the ciphertext.
In a known-plaintext attack, the attacker knows some pairs (x',y'), (x",y"), .. of plaintexts/ciphertexts (e.g., Hill ciphers), the limit in this case is that it is a linear transformation of plaintext block into a cipher block.

A chosen-plaintext attack (CPA) is an attack model for cryptoanalysis which presumes that the attacker can ask and obtain the ciphertexts for given plaintexts.
A chosen-ciphertext attack (CCA) is an attack model for cryptoanalysis where the cryptoanalyst can gather information by obtaining the decryption of chosen ciphertexts.

Exercise 2. Encrypt and decrypt message (1,3) using a Hill cipher with K $= \begin{bmatrix} 1 & 2 \\ 4 & 3 \end{bmatrix}$

Encryption

E$_K$(1,3) $= (1,3) \times \begin{bmatrix} 1 & 2 \\ 4 & 3 \end{bmatrix}$ mod26 $= (1 \times 1 + 3 \times 4, 1 \times 2 + 3 \times 3)$ mod26 $= (1 + 12, 2 + 9)$ mod26 $= (13, 11)$ mod26 $= (13, 11)$.

Decryption

K^{-1} =det^{-1}(K) $\begin{bmatrix} 3 & -2 \\ -4 & 1 \end{bmatrix}$ mod26 $=$ det^{-1}(K)$\begin{bmatrix} 3 & 24 \\ 22 & 1 \end{bmatrix}$ mod26.
det(K) $= (1 \times 3 - 2 \times 4)$mod26 $= (3 - 8)$mod26 $=$ -5 mod26 $=$ 21.

$\det^{-1}(K) = 5$, $21 \times 5 \bmod 26 = 105 \bmod 26 = 1$.
$K^{-1} = \det^{-1}(K)\begin{bmatrix}3 & 24\\22 & 1\end{bmatrix} \bmod 26 = 5\begin{bmatrix}3 & 24\\22 & 1\end{bmatrix} \bmod 26 = \begin{bmatrix}15 & 120\\110 & 5\end{bmatrix} \bmod 26 = \begin{bmatrix}15 & 16\\6 & 5\end{bmatrix}$.
$D_K(13{,}11) = (13{,}11)\begin{bmatrix}15 & 16\\6 & 5\end{bmatrix} = (13 \times 15 + 11 \times 6, 13 \times 16 + 11 \times 5) \bmod 26 = (261, 263)$ $\bmod 26 = (1, 3)$.

Attack to the Hill cipher

If I am an attacker and I have a specific number of pairs (plaintext, ciphertext), I can attack the Hill cipher. With a m×m matrix, I need m pairs, if I have a 2×2 matrix, I will need 2 pairs (plaintext, ciphertext) to be able to extract the key. Once I have the key, I can decrypt all the messages that will be sent later on.

We know that:

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

which can be written as $Y = XK \bmod 26$ with:

$$X = \begin{bmatrix}x_1^1 & .. & x_m^1\\ .. & .. & ..\\ x_1^m & .. & x_m^m\end{bmatrix} \text{ and } Y = \begin{bmatrix}y_1^1 & .. & y_m^1\\ .. & .. & ..\\ y_1^m & .. & y_m^m\end{bmatrix}$$

If X^{-1} exists, we can write $X^{-1}Y \bmod 26 = X^{-1}XK \bmod 26$ and then $K = X^{-1}Y \bmod 26$.

Example 21. Let's define:

$$x_1=(5{,}9) \rightarrow y_1=(19{,}4)$$
$$x_2=(2{,}5) \rightarrow y_2=(24{,}11)$$

We can write $(19{,}4) = (5{,}9)K \bmod 26$ and $(24{,}11) = (2{,}5)K \bmod 26$.
We can construct X and Y matrices as follows:

$$X = \begin{bmatrix}5 & 9\\2 & 5\end{bmatrix} \text{ and } Y = \begin{bmatrix}19 & 4\\24 & 11\end{bmatrix}.$$

Now, if X^{-1} exists, we can recover the key K. We compute X^{-1} in the same way we did previously for K^{-1}.
$X^{-1} = \det^{-1}(X)\begin{bmatrix}5 & -9\\-2 & 5\end{bmatrix} \bmod 26 = \det^{-1}(X)\begin{bmatrix}5 & 17\\24 & 5\end{bmatrix} \bmod 26$.

$\det(X) = (5 \times 5 - 9 \times 2) = (25\text{-}18) = 7$.
$\det^{-1}(X)$ is a number a such that $7 \times a = 1 \bmod 26$, this number $a = 15$.

Thus $X^{-1} = \det^{-1}(X)\begin{bmatrix}5 & 17\\24 & 5\end{bmatrix} \bmod 26 = 15\begin{bmatrix}5 & 17\\24 & 5\end{bmatrix} \bmod 26 = \begin{bmatrix}75 & 255\\360 & 75\end{bmatrix} \bmod 26 =$ $\begin{bmatrix}23 & 21\\22 & 23\end{bmatrix}$.

Then, $\mathrm{K} = \mathrm{X}^{-1}\mathrm{Y} \bmod 26 = \begin{bmatrix} 23 & 21 \\ 22 & 23 \end{bmatrix} \begin{bmatrix} 19 & 4 \\ 24 & 11 \end{bmatrix} \mathrm{mod} 26 = \begin{bmatrix} 23 \times 19 + 21 \times 24 & 23 \times 4 + 21 \times 11 \\ 22 \times 19 + 23 \times 24 & 22 \times 4 + 23 \times 11 \end{bmatrix}$ $\mathrm{mod} 26 = \begin{bmatrix} 941 & 323 \\ 970 & 341 \end{bmatrix} \bmod 26 = \begin{bmatrix} 5 & 11 \\ 8 & 3 \end{bmatrix}$.

So, our key $\mathrm{K} = \begin{bmatrix} 5 & 11 \\ 8 & 3 \end{bmatrix}$.

Modern ciphers always contain a non-linear component to prevent this kind of attacks.

If th matrix X is not invertible, if the attacker has no more pairs of (plaintext, ciphertext) he lose, otherwise he can try constructing other X matrices that maybe are invertible.

Block ciphers

Block ciphers are cryptosystems that "reuse" the same key to encrypt letters or blocks of the plaintext. For example shift ciphers are part of block ciphers. They are weak because if an attacker know that it is always used the same key, he could recover the key and decrypt al the messages.

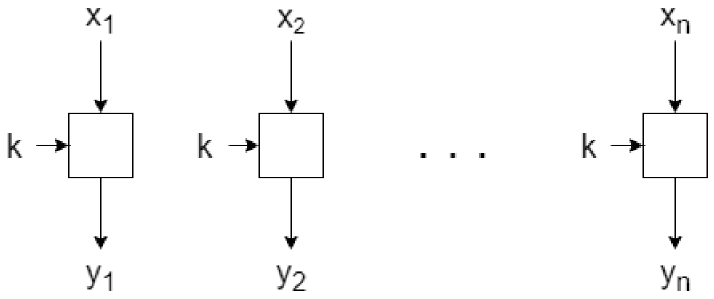


Figure 11: Example of block ciphers.

Stream ciphers

Definition 10. **Stream ciphers** are cryptosystems that use a stream of key $z_1, z_2, ..., z_n$ instead of a single one.

The idea is to encrypt the first letter of the plaintext with z_1, the second with z_2 and so on, it doesn't matter if we encrypt a letter or a block of text, what matters is that the key used is always different.

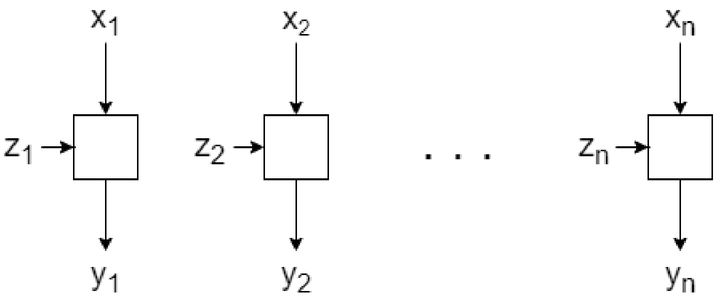


Figure 12: Example of stream ciphers.

Having a different key for each letter or block of the plaintext is of course appealing but not much practical. The stream of keys is thus usually derived starting from an initial key k, but it can also depend on previous parts of the plaintext. In general we say that

$$z_i = f_i(k, x_i, ..., x_{i-1}).$$

That means that the i-th key depends on k and on the previous i-1 letters or blocks. So $z_1 = f_1(k)$, we can compute it without knowledge of the plaintext, to compute z_2, instead, we need to know x_1 because $z_2 = f_2(k,x_1)$. The values are thus computed in the following sequence: z_1, x_1, z_2, x_2, ...
This slow down a bit the procedure because I can't do anything in parallel.

Block ciphers are a subset of stream ciphers in which i have z_i=k for each i.

A stream cipher is **periodic** if its key stream has the form:

$$z_1,\ z_2,\ ..,\ z_d, z_1,\ z_2,\ ..,\ z_d, z_1,\ ...$$

So if it repeats after **d** steps. This form of cipher reminds us to the **Vigenére cipher**. It can be seen as a stream cipher acting on single letters and with a periodic key stream.

Exercise 3. Formalize the cipher giving (P,C,K,E,D) and defining the key stream z_i.

- P=C=K=Z_{26};
- $E_{zi}(x_i) = (x_i + z_i)$ mod26;
- $D_{zi}(y_i) = (y_i - z_i)$ mod26;
- $z_i = k_{(i \bmod m)}$.

A stream cipher is **synchronous** if its key stream does not depend on the plaintexts (for example $z_i = f_i(k)$ for all i). So the key stream can be generated starting from k and independently on the plaintext. It is useful to improve efficiency because we do not need to obtain x_i to compute z_{i+1}, so the key stream can be generated offline, before the actual ciphertext is received. We can consider Caesar cipher Hill cipher and also Vigenére cipher part of synchronous ciphers.

Asynchronous stream ciphers

In general asynchronous stream ciphers generate keys that depends either from k and from the previous plaintexts:

$$z_i = f_i(k, x_1, ..., xi - 1)$$

This means that, if we are decrypting, we need to decrypt and, at the same time, compute the keys stream as a key can depend on previous plaintexts. An example is the **Autokey cipher**. We define it in the same way of a shift cipher, so P=C=K=Z_{26} and

- $E_{zi}(x_i) = (x_i + z_i)$ mod26;
- $D_{zi}(y_i) = (y_i - z_i)$ mod26.

We define the key stream as:

$$z_i = \begin{cases} k & \text{if i=0} \\ x_{i-1} & \text{if i}\geq 2 \end{cases}$$

The first key is the initial key k and the next keys are the same as the previous plaintext.

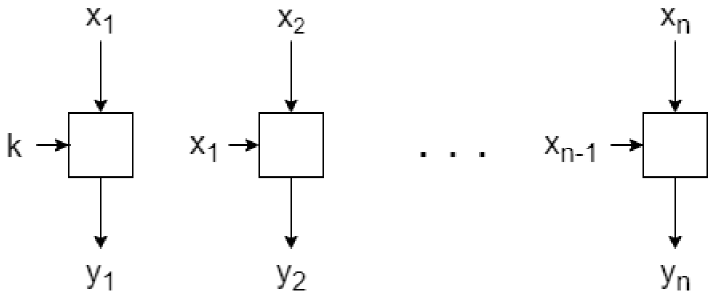


Figure 13: Encryption in an autokey cipher.

Exercise 4. Let's assume the plaintext is the word "networksecurity", what is the encryption using k=5?
First of all we need to substitute our word with the corresponding position in the alphabet and it is: "13 4 19 22 14 17 10 18 4 2 20 17 8 19 24". We will encrypt our string as follows:

E(13) = (13+5) mod26 = 18
E(4) = (4+13) mod26 = 17
...
E(24) = (24+19) mod26 = 17

At this point we just need to transform back our numbers into letters and we obtain the ciphertext.

Exercise 5. Try to extract the plaintext from this word encoded using Autokey. We do not know the key k.

```
FTPNIH
```

Since we do not know the key k, we have to try all the 26 possible k to see which decryption produces a meaningful word.

- k=0 will produce `FOBMWL`, nonsense;
- k=1 will produce `EPANVM`, nonsense;
- k=2 will produce `DQZOUN`, nonsense;
- k=3 will produce `CRYPTO`, it has sense!

Exercise 6. Suppose that we know that `FRIDAY` has been encrypted as `PQCFKU` using the Hill cipher, What is K? Assume K is a 2×2 matrix.

We can consider the pairs:

```
(F,R) → (P,Q) = (5,17) → (15,16)
(I,D) → (C,F) = (8,3) → (2,5)
(A,Y) → (K,U) = (0,24) → (10,20)
```

We can construct X and Y matrices as follows:

$$\mathrm{X} = \begin{bmatrix} 5 & 17 \\ 8 & 3 \end{bmatrix} \text{ and } \mathrm{Y} = \begin{bmatrix} 15 & 16 \\ 2 & 5 \end{bmatrix}.$$

Now, if X^{-1} exists, we can recover the key K. We compute X^{-1} in the same way we did previously for K^{-1}.

$\mathrm{X}^{-1} = \det^{-1}(\mathrm{X}) \begin{bmatrix} 3 & -17 \\ -8 & 5 \end{bmatrix} \mathrm{mod}26 = \det^{-1}(\mathrm{X}) \begin{bmatrix} 3 & 9 \\ 18 & 5 \end{bmatrix} \mathrm{mod}26.$

$\det(\mathrm{X}) = (5 \times 3 - 17 \times 8) =$ (15-136) = -121 mod26 = -17 mod26 = 9.

$\det^{-1}(\mathrm{X})$ is a number a such that $9 \times a = 1$ mod26, this number is $a = 3$.

Thus $\mathrm{X}^{-1} = \det^{-1}(\mathrm{X}) \begin{bmatrix} 3 & 9 \\ 18 & 5 \end{bmatrix} \mathrm{mod}26 = 3 \begin{bmatrix} 3 & 9 \\ 18 & 5 \end{bmatrix} \mathrm{mod}26 = \begin{bmatrix} 9 & 27 \\ 54 & 15 \end{bmatrix} \mathrm{mod}26 = \begin{bmatrix} 9 & 1 \\ 2 & 15 \end{bmatrix}.$

Then, $\mathrm{K} = \mathrm{X}^{-1}\mathrm{Y} \bmod 26 = \begin{bmatrix} 9 & 1 \\ 2 & 15 \end{bmatrix} \begin{bmatrix} 15 & 16 \\ 2 & 5 \end{bmatrix} \mathrm{mod}26 = \begin{bmatrix} 9 \times 15 + 1 \times 2 & 9 \times 16 + 1 \times 5 \\ 2 \times 15 + 15 \times 2 & 2 \times 16 + 15 \times 5 \end{bmatrix}$
$\mathrm{mod}26 = \begin{bmatrix} 137 & 149 \\ 60 & 107 \end{bmatrix} \bmod 26 = \begin{bmatrix} 7 & 19 \\ 8 & 3 \end{bmatrix}.$

So, our key $\mathrm{K} = \begin{bmatrix} 7 & 19 \\ 8 & 3 \end{bmatrix}.$

Lecture 5

Definition 11. A **perfect cipher** is a cipher that can never be broken, even after an unlimited time.

But do perfect ciphers exist? Yes, but they are only theoretically (they can be implemented), in practice they are unpractical.

The theory behind perfect ciphers has been developed by **Claude Shannon**. It assumes an only-ciphertext model of the attacker, this means that the attacker only knows the ciphertext y and tries to find the plaintext x or the key k.

We can provide another definition for perfect ciphers:

Definition 12. 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.

To formalize them we need to introduce a bit of probability:

- p$_p$(x) is the probability of plaintext x to occur;
- p$_K$(k) is the probability of certain key k to be used as encryption key.

Given a plaintext and a key there exists a unique corresponding ciphertext.

The two probability distributions p$_p$(x) and p$_K$(k) induce a probability distribution on the ciphertexts.

$$p_c(y) = \sum_{k \in K, \exists x | E_k(x)=y} p_K(k) \times p_p(D_k(y)) \tag{1}$$

This means that, given a ciphertext y, we look for all the keys that can give such a ciphertext from some plaintext x. Then we sum the probability of all such keys times the probability of the corresponding plaintext.

Example 22. Let's consider the following toy-cipher with P=a,b, K=k$_1$,k$_2$, C1,2,3. The encryption is defined in the following table:

E	**a**	**b**
k$_1$	1	2
k$_2$	2	3

Let p$_p$(a)=3/4, p$_p$(b)=1/4, p$_K$(k$_1$)=p$_K$(k$_2$)=1/2.

Now we want to compute p$_C$(1), p$_C$(2), p$_C$(3).

p$_C$(1) = p$_p$(a) × p$_K$(k$_1$) = 3/4 × 1/2 = 3/8
p$_C$(2) = p$_p$(a) × p$_K$(k$_2$) + p$_p$(b) × p$_K$(k$_1$) = 3/4 × 1/2 + 1/4 × 1/2 = 3/8 + 1/8 = 1/2
p$_C$(3) = p$_p$(b) × p$_K$(k$_2$) = 1/4 × 1/2 = 1/8

Definition 13. The **conditional probability of a ciphertext** y with respect to a plaintext x, computes how likely is a certain y once we fix x.

$$p_c(y|x) = \sum_{k \in K, E_K(x)=y} p_K(x) \tag{1}$$

More simply is just the sum of the probability of all keys giving y from x.

Example 23. The conditional probability of ciphertext 1 with respect to the two plaintexts a and b is:

$$p_c(1|a) = \sum_{k \in K, E_K(x)=y} p_K(x) = p_K(k_1) = 1/2 \tag{1}$$

$$p_c(1|b) = \sum_{k \in K, E_K(x)=y} p_K(x) = 0 \tag{2}$$

We can notice that 1 can never be obtained from b. Furthermore we can see that if an attacker sees a 1, he can deduce that it is an a, this means that this is not a perfect cipher.

The conditional probability of a plaintext with respect to a ciphertext is related to the security of the cipher. 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).

With Pr(y)>0, we can define the **Bayes' Theorem** as:

$$Pr(x|y) = \frac{Pr(x) \times Pr(y|x)}{Pr(y)} \tag{1}$$

We can apply this theorem to our case to get the conditional probability of a plaintext x with respect to a ciphertext y:

$$p_p(x|y) = \frac{p_p(x) \times p_c(y|x)}{p_c(y)} \tag{1}$$

Example 24. At this point we can compute p$_p$(a|1) and p$_p$(b|1) as follows:

$$p_p(a|1) = \frac{p_p(a) \times p_c(1|a)}{p_c(1)} = \frac{3/4 \times 1/2}{3/8} = 1 \tag{1}$$

$$p_p(b|1) = \frac{p_p(b) \times p_c(1|b)}{p_c(1)} = \frac{1/4 \times 0}{3/8} = 0 \tag{2}$$

Thus, when observing a 1 as a result of a conditional probability of a plaintext x with respect to a ciphertext y, we are sure it is plaintext x, and this means that the cipher is completely insecure.

Exercise 7. What will be the probabilities of plaintexts a and b with respect to ciphertext 3?

First of all we need to find the conditional probabilities of ciphertext 3 with respect to the plaintexts a and b:

$$p_c(3|a) = \sum_{k \in K, E_K(x)=y} p_K(x) = 0 \tag{1}$$

$$p_c(3|b) = \sum_{k \in K, E_K(x)=y} p_K(x) = p_K(k_2) = 1/2 \tag{2}$$

We can now compute the conditional probabilities of plaintexts a and b with respect to the ciphertext 3:

$$p_p(a|3) = \frac{p_p(a) \times p_c(3|a)}{p_c(3)} = \frac{3/4 \times 0}{1/8} = 0 \tag{1}$$

$$p_p(b|3) = \frac{p_p(b) \times p_c(3|b)}{p_c(3)} = \frac{1/4 \times 1/2}{1/8} = 1 \tag{2}$$

Exercise 8. What if we compute the probabilities of a and b when observing 2?

First of all we need to find the conditional probabilities of ciphertext 2 with respect to the plaintexts a and b:

$$p_c(2|a) = \sum_{k \in K, E_K(x)=y} p_K(x) = p_K(k_1) = 1/2 \tag{1}$$

$$p_c(2|b) = \sum_{k \in K, E_K(x)=y} p_K(x) = p_K(k_2) = 1/2 \tag{2}$$

We can now compute the conditional probabilities of plaintexts a and b with respect to the ciphertext 2:

$$p_p(a|2) = \frac{p_p(a) \times p_c(2|a)}{p_c(2)} = \frac{3/4 \times 1/2}{1/2} = 3/4 \tag{1}$$

$$p_p(b|2) = \frac{p_p(b) \times p_c(2|b)}{p_c(2)} = \frac{1/4 \times 1/2}{1/2} = 1/4 \tag{2}$$

At this point we can provide an extra definition for perfect ciphers:

Definition 14. A cipher is perfect iff $\mathrm{p}_p(\mathrm{x}|\mathrm{y})=\mathrm{p}_p(\mathrm{x})$ for all x in P and for all y in 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.

A cipher is perfect if there is some key that maps any message to any ciphertext with equal probability.

Lecture 6

To prove that a cipher is not perfect, we just need to find a x in P and a y in C that does not satisfy definition 14

Exercise 9. Prove that the shift cipher with $\mathrm{p}_K(\mathrm{k}) = 1/|\mathrm{K}| = 1/26$ (i.e., with keys picked at random for each letter of the plaintext), is a perfect cipher. If we change key at any time we encrypt a letter, the shift cipher becomes perfect (unbreakable).

We can define the cipher as follows:

- $\mathrm{P}=\mathrm{C}=\mathrm{K}=\mathrm{Z}_{26}$;
- $\mathrm{E}_k(\mathrm{x}) = (\mathrm{x}+\mathrm{k})$ mod26;
- $\mathrm{D}_k(\mathrm{y}) = (\mathrm{y}\text{-}\mathrm{k})$ mod26;
- k changes at each encryption.

We want to prove that definition 14 holds for this cipher.

We compute the probability of a generic ciphertext y as:

$$\begin{aligned} p_C(y) &= \sum_{k \in K, \exists x. E_k(x)=y} p_K(k) \times p_p(D_k(y)) && (1) \\ &= \frac{1}{26} \sum_{k \in K, \exists x. E_k(x)=y} p_p(D_k(y)) && (2) \\ &= \frac{1}{26} \sum_{k \in K} p_p((y-k) mod 26) && (3) \\ &= \frac{1}{26} \sum_{x \in P} p_p(x) = \frac{1}{26} && (4) \end{aligned}$$

Note that for each key k, we always have the plaintext (y-k) mod26 that gives y when encrypted under k, and that for all possible keys gives all possible plaintexts x and sums to 1.

$$\begin{aligned} p_c(y|x) &= \sum_{k \in K, E_k(x)=y} p_K(k) && (1) \\ &= p_K((y-x) mod 26) = \frac{1}{26} && (2) \end{aligned}$$

We get that k = (y-x) mod26 from the decryption formula, given x and y, there exists a unique key k that encrypts x as y and it is (y-x)mod26.

Finally we can say that:

$$p_p(x|y) = \frac{p_p(x) \times p_c(y|x)}{p_c(y)} \tag{1}$$
$$= \frac{p_p(x) \times \frac{1}{26}}{\frac{1}{26}} = p_p(x) \tag{2}$$

We have demonstrated that $p_p(y|x) = p_p(x)$ for all x in P and for all y in C, thus the cipher is perfect.

Theorem 1. Let $p_c(y) > 0$ for all y. A cipher is perfect only if |K|≥|P|.

This means that a necessary condition for a cipher to be perfect is that the number of keys is at least the same as the number of plaintexts.

Proof. For Bayes' theorem:

$$p_p(x|y) = \frac{p_p(x) \times p_c(y|x)}{p_c(y)}$$

Thus, given $p_p(x|y) = p_p(x)$ (the cipher is perfect), we have $p_c(y|x) = p_c(y)$ (these two quantities erase) for all x in P and for all y in C.

As an assumption $p_c(y) > 0$.
If we fix x, we obtain that for each y, $p_c(y|x) = p_c(y) > 0$. This means that there exists at least one key k such that E$_k$(x)=y (otherwise $p_c(y|x) = 0$).

All such keys are different since E$_k$ is a function and we have fixed x, and x cannot be mapped to two different ciphertexts by the same key. Thus, we have at least one key for each ciphertext (|K|≥|C|).

Since, for any cipher (not necessarily perfect), E$_k$ injects the set of plaintexts into the set of ciphertext, we also have |C|≥|P|, thus |K|≥|C|≥|P|, i.e.,

$$\mathbf{|K|\geq|P|}$$

■

Exercise 10. Prove that the ciphertext defined as E$_k$(x$_1$,...,x$_d$)=(x$_1$+k,...,x$_d$+k)mod26 is not perfect.

We can prove this in two ways:

1. using the theorem 1, proving that |K|<|P|.
   Our set K={0, 1, 2, .., 25}, so |K|=26, we need to find |P|. Since x={x$_1$,..,x$_d$} and each x$_i$ can assume 26 values, we have that |P|=26^d. Thus 26<26^d, so |K|<|P| (with d≥2).

2. using definition 14. Let's assume x={x$_1$,..,x$_d$} is an English word and let's also assume d=5. We are looking for an English word of length 5, for example *GOOFY*. Since it is a real word, p_p(GOOFY)>0. We want to find a y such that p_p(GOOFY|y)≠ p_p(GOOFY)>0. For example with y=AAAAA, p_p(GOOFY|AAAAA)=0 and we are done. We could have used other y, such as y=ABCDE.

***Theorem* 2.** Let |P|=|C|=|K|. A cipher is perfect iff

1. $p_K(k) = 1/|K|$ for all k in K;
2. for each x in P and y in C, there exists exactly one key k such that E$_k$(x)=y.

The theorem states that, for a cipher to be perfect (given that the size of P, C and K is the same), keys should be picked at random for any encryption and each plaintext is mapped into each ciphertext through a unique key.

Proof. We will prove the → direction of the theorem.

For theorem 1, in a perfect cipher |K|≥|C| if p_c(y)>0 for all y. If we fix x, we obtain that for each y p_c(y|x)=p_c(y)>0, i.e., there exists at least one key k such that E$_k$(x)=y and all of the other keys are different.

Here (by assumption) |K|=|C|, meaning that all of these keys k are unique (otherwise we would have |K|>|C|). Since this holds for each x and y, we have proved condition 2, i.e., that for each x in P and y in C, there exists exactly one key k such that E$_k$(x)=y.

To prove condition 1, it is enough to notice that given

$$p_c(y|x) = \sum_{k \in K, E_k(x)=y} p_K(k)$$

p_c(y|x)=p_p(y), 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_K(k)=p_c(y|x)=p_c(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_K(k)=p_c(y). Since the number of keys is |K|, and since they all have the same probability (p_c(y)), and given that the sum of the probabilities of all keys must be 1 and |P|=|C|=|K|, we obtain condition 1, p_K(k)=1/|K|.

■

We can use this theorem to prove that a cipher is (or is not) perfect.

The one time pad

This perfect cipher has been used for the telegraph and is a binary variant of Vigenére with keys picked at random.

More precisely we have P=C=K=Z$_{2^d}$ with p_K(k)=1/|K|=1/2^d for all k in K. 2^d means sequences of binary numbers of length d.

The encryption function is the following: E$_{k_1,...,k_d}$(x$_1$,...,x$_d$) = (x$_1$ xor k$_1$,...,x$_d$ xor k$_d$), where xor is a bitwise xor operation.

Is it a perfect cipher? Condition 2 of theorem 2 is satisfied by definition. Condition 2 holds too because we are using xor operations.

Lesson learned

Shannon theory on perfect ciphers shows that they 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 only once. Once the book has been used completely, new shared books were necessary.

Lecture 7

Exercise 11. Consider the following cipher with P={a,b}, K={k$_1$,k$_2$,k$_3$}, C={1,2,3,4}. Encryption is defined by the following table:

	a	**b**
k$_1$	1	2
k$_2$	2	3
k$_3$	3	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).

First of all let's compute all the p_c for 1,2,3,4:

$$p_c(1) = p_p(a) \times p_K(k_1) = 1/4 \times 1/2 = 1/8$$

$$p_c(2) = p_p(b) \times p_K(k_1) + p_p(a) \times p_K(k_2) = 3/4 \times 1/2 + 1/4 \times 1/4 = 7/16$$

$$p_c(3) = p_p(b) \times p_K(k_2) + p_p(a) \times p_K(k_3) = 3/4 \times 1/4 + 1/4 \times 1/4 = 1/4$$

$$p_c(4) = p_p(b) \times p_K(k_3) = 3/4 \times 1/4 = 3/16$$

Then we can compute

$$p_c(1|a) = p_K(k_1) = 1/2$$

$$p_c(2|a) = p_K(k_2) = 1/4$$

$$p_c(3|a) = p_K(k_3) = 1/4$$

$$p_c(4|a) = 0$$

$$p_c(1|b) = 0$$

$$p_c(2|b) = p_K(k_1) = 1/2$$

$$p_c(3|b) = p_K(k_2) = 1/4$$

$$p_c(4|b) = p_K(k_3) = 1/4$$

Finally:

$$p_p(a|1) = \frac{p_c(1|a) \times p_p(a)}{p_c(1)} = \frac{1/2 \times 1/4}{1/8} = 1$$

$$p_p(a|2) = \frac{p_c(2|a) \times p_p(a)}{p_c(2)} = \frac{1/4 \times 1/4}{7/16} = 1/7$$

$$p_p(a|3) = \frac{p_c(3|a) \times p_p(a)}{p_c(3)} = \frac{1/4 \times 1/4}{1/4} = 1/4$$

$$p_p(a|4) = \frac{p_c(4|a) \times p_p(a)}{p_c(4)} = \frac{0 \times 1/4}{3/16} = 0$$

$$p_p(b|1) = \frac{p_c(1|b) \times p_p(b)}{p_c(1)} = \frac{0 \times 3/4}{1/8} = 0$$

$$p_p(b|2) = \frac{p_c(2|b) \times p_p(b)}{p_c(2)} = \frac{1/2 \times 3/4}{7/16} = 6/7$$

$$p_p(b|3) = \frac{p_c(3|b) \times p_p(b)}{p_c(3)} = \frac{1/4 \times 3/4}{1/4} = 3/4$$

$$p_p(b|4) = \frac{p_c(4|b) \times p_p(b)}{p_c(4)} = \frac{1/4 \times 3/4}{3/16} = 1$$

Composition of ciphers

A one-time-pad provides absolute security in theory, but difficult to implement in practice. Modern ciphers are based on very simple operations, such as substitution, xor, etc., that are combined in a smart way so to make the overall algorithm strong and really hard to cryptoanalyse.

Combining simple ciphers does not always improve security. Let's consider the shift cipher composed twice: we first shift by k_1 and then by k_2 modulo 26. But this is equivalent to shift by k_1+k_2 modulo 26.

Example 25. Shift of 3 and then of 2:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
DEFGHIJKLMNOPQRSTUVWXYZABC
FGHIJKLMNOPQRSTUVWXYZABCDE
```

but this is equivalent to a shift of 5:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
FGHIJKLMNOPQRSTUVWXYZABCDE
```

Definition 15. **Composition:** we consider two ciphers S^1=(P^1, C^1, K_1, E^1, D^1) and S^2=(P^2, C^2, K_2, E^2, D^2).

We let P^1=C^1=P^2=C^2, that we note as P and 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 the composition as $S^1 \times S^2$=(P, C, $K_1 \times K_2$, E, D) with:

- $E_{k_1,k_2}(x) = E^2_{k_2}(E^1_{k_1}(x))$;
- $D_{k_1,k_2}(y) = D^1_{k_1}(D^2_{k_2}(y))$;

Exercise 12. Formally apply the composition of the two shift ciphers previously mentioned.

Formally we have $\mathrm{E}^1_k(\mathrm{x})=\mathrm{E}^2_k(\mathrm{x})=(\mathrm{x}+\mathrm{k})\mathrm{mod}26$
and
$\mathrm{E}_{(k_1,k_2)}(\mathrm{x}) = \mathrm{E}^2_{k_2}(\mathrm{E}^1_{k_1}(\mathrm{x})) = ((\mathrm{x}+\mathrm{k}_1)\ \mathrm{mod}26+\mathrm{k}_2)\ \mathrm{mod}26 = (\mathrm{x}+(\mathrm{k}_1+\mathrm{k}_2)\mathrm{mod}26)\mathrm{mod}26 = \mathrm{E}^1_{k_1+k_2 mod26}(\mathrm{x})$

This proves that composing the shift cipher twice is equivalent to applying it once using as a key the sum of the two keys k_1 and k_2.

Exercise 13. 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?

Let's consider an example of substitution:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
VCFIORBMPLYUQXJHNDWSKETZAG
```

If we shift of 5 (for example) and then we substitute it becomes:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
RBMPLYUQXJHNDWSKETZAGVCFIO
```

This is nothing more then a substitution cipher with a different key. To decrypt we need to apply first the substitution and then the shift back.

If we do the opposite, so first we substitute and then we shift we would have something similar to the following example:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
VCFIORBMPLYUQXJHNDWSKETZAG
AHKNTWGRUQDZVCOMSIBXPJYEFL
```

Once again it corresponds to a substitution cipher with a different key. Of course to decrypt we would have first to shift back and then substitute.

Exercise 14. Consider the composition of a 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?

The Vigenère cipher with key `ALICE` encryption can be represented as:

$$\mathrm{E}_{k_1,...,k_m}(\mathrm{x}_1,...,\mathrm{x}_m) = (\mathrm{x}_1+0,\ \mathrm{x}_2+11,\ \mathrm{x}_3+8,\ \mathrm{x}_4+2,\ \mathrm{x}_5+4,\ ..)\ \mathrm{mod}26$$

A shift cipher with key 8 can be represented as:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
IJKLMNOPQRSTUVWXYZABCDEFGH
```

Then the composition is: $\mathrm{E}_{k_1,k_2}(\mathrm{x}) = (\mathrm{x}_1+0+8,\ \mathrm{x}_2+11+8,\ \mathrm{x}_3+8+8,\ \mathrm{x}_4+2+8,\ \mathrm{x}_5+4+8,\ ..)\ \mathrm{mod}26 = (\mathrm{x}_1+8,\ \mathrm{x}_2+19,\ \mathrm{x}_3+16,\ \mathrm{x}_4+10,\ \mathrm{x}_5+12,\ ..)\ \mathrm{mod}26$

This is nothing more then a Vigenére cipher with key `ITQKM`

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 is said to be **idempotent**, written S×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. Almost any modern cipher repeats a basic core of operations for a certain number of rounds (iteration). 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.

Proof. Consider one iteration of their composition

$$(S^1 \times S^2) \times (S^1 \times S^2) \tag{1}$$

$$= S^1 \times (S^2 \times S^1) \times S^2 \qquad \text{associative property} \tag{2}$$

$$= S^1 \times (S^1 \times S^2) \times S^2 \qquad \text{commutative property} \tag{3}$$

$$= (S^1 \times S^1) \times (S^2 \times S^2) \qquad \text{associative property} \tag{4}$$

$$= S^1 \times S^2 \qquad \text{idempotence of the initial ciphers} \tag{5}$$

■

Exercise 15. Apply the above result to show that the composition of Vigenére cipher S^1 and the shift cipher S^2 is idempotent.

The composition is a Vigenére cipher S^3 with key k_1+k_2 mod26, ..., k_d+k_2 mod26, $S^1 \times S^2 = S^3$.

Thus $(S^1 \times S^2) \times (S^1 \times S^2) = S^3 \times S^3 = S^3$ which is again a Vigenére cipher with a new key. This holds iterating many times.

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.

The Advanced Encryption Standard (AES) cipher

AES has been selected by the National Institute of Standards and Technology (NIST) after a five-year long competition. The original name is Rijndael from the names of the two inventors, Joan Daemen and Vincent Rijmen.

As any modern cipher, AES is the composition of simple operations and contains a nonlinear component to avoid known-plaintexts attacks. The composed operation give a non-idempotent cipher that is iterated for a fixed number of rounds.

It has been selected since it provides:

- high security guarantees;
- high performance;
- flexibility (different key length)

All of these features are, in fact, crucial for any modern cipher.

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 (168 bits). In fact, DES key of only 56 bits(64-8 control bits) is too short to resist brute-forcing on modern, parallel computers.

Mathematical background of AES

AES works on the Galois Field with 2^8 elements, noted $GF(2^8)$. Intuitively it is the set of all 8-bits digits with sum and multiplication performed by interpreting the bits as (binary) coefficients of polynomials. For example 11010011 can be seen as $x^7+x^6+x^4+x+1$ and 00111010 is $x^5+x^4+x^3+x$.

The sum of $x^7+x^6+x^4+x+1$ and $x^5+x^4+x^3+x$ will thus be $x^7+x^6+x^5+x^3+1$ since two 1's coefficient becomes 0 modulo 2=0, and the term disappears. We see that sum and subtraction are just the bit-wise XOR of the binary numbers, 11010011 XOR 00111010 = 11101001 which is $x^7+x^6+x^5+x^3+1$.

The product is done modulo the irreducible polynomial $x^8+x^4+x^3+x+1$, irreducible means that it cannot be written as the product of two other polynomials, it is, intuitively, the equivalent of primality.

Example 26. $(x^7+x^6+x^4+x+1)\times(x^5+x^4+x^3+x) = x^{12} + x^{11} + x^{10} + x^8 + x^{11} + x^{10} + x^9 + x^7 + x^9 + x^8 + x^7 + x^5 + x^6 + x^5 + x^4 + x^2 + x^5 + x^4 + x^3 + x = x^{12} + x^6 + x^5 + x^3 + x^2 + x$

Now we need 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 reminder. $x^{12}/x^8=x^4$, thus $(x^8+x^4+x^3+x+1)*x^4 = x^{12}+x^8+x^7+x^5+x^4$.

At this point we need to subtract this polynomial (to get the remainder), we obtain $x^8 + x^7 + x^6 + x^4 + x^3 + x^2 + x$ and we repeat the process obtain the final remainder: $x^7 + x^6 + x^2 + 1$.

Exercise 16. Multiply $(x^4+x^3+1)\times(x^5+1)$

$= x^9+x^4+x^8+x^3+x^5+1 = x^9+x^8+x^5+x^4+x^3+1$

We divide it by the irreducible polynomial: $x^9/x^8 = x$, thus $(x^8+x^4+x^3+x+1)*x = x^9+x^5+x^4+x^2+x$.

We subtract this polynomial from the result of the multiplication to obtain the reminder: $x^8+x^3+x^2+x+1$.

We repeat the process and we obtain the final reminder x^4+x^2.

Lecture 8

An example of optimization on numbers (on polynomial we will use XOR instead of SUM) is the following:

Example 27. Let's multiplicate 11 and 1011.

a	**b**	**p**
11	1011	0
110	101	$0\oplus 11=11$
1100	10	$11\oplus 110=101$
11000	1	101
110000		$101\oplus 11000 = 11101$

Once we have an empty space in the **b** column, we will have the result of the multiplication in the **p** column.

The method seen at the end of the last lecture is, on average, quadratic with respect to the number of bits (8). Instead, the method just seen takes a number of steps equals to the length of the number in the **b** column.

```
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               #shift b to the right
    hbit = (a&0x80)!=0  #true if the most significant bit of a is set
    a<<=1               #shift 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-bits number
  return p
```

This method works only if the resulting number smaller than x^7.

To test the above code it is sufficient to open a python shell (by running python from the terminal) and copy-paste the function into the shell, giving enter at the end. `AESmult(0b11010011,0b00111010)` will produce 197 as result, `bin (AESmult (0b11010011, 0b00111010))` will produce 0b11000101.

The correctness of the optimization of the product derives from invariant after each loop:

- ab+p is the product of the initial a and b (all operations are done in the Galois Field);
- since b is 0 at the end, we have that p contains the product.

Some observations:

- 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;
- if b is odd, the polynomial is not divisible by x, so we throw away the least significant bit (this is what the right shift does) and we accumulate one a in p to compensate and keep the invariant;
- when a becomes more than 2^8, we need to sum to it the modulus 100011011 (the irreducible polynomial), i.e. 0x11b, to keep it 8-bits long.

The AES cipher

Advanced Encryption Standard (AES) is a symmetric key algorithm that uses the same key for both encryption and decryption processes, it 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,...,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-bits keys, 12 rounds for 192-bits keys and 14 rounds for 256-bits 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:

- bitwise xor;
- fixed non-linear substitution;
- shifting of matrix rows;
- matrix column multiplications.

AddRoundKey

The round key is bitwise xor-ed with the block. A round key is thus 128 bits, independently of the chosen key size.

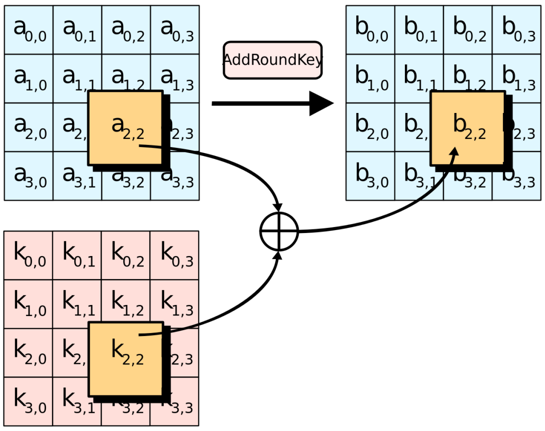


Figure 14: AddRoundKey.

SubBytes

A fixed non-linear substitution, called S-box, is applied to each byte of the block. Given a byte in hexadecimal notation, the first digit is used to select a row and the second one to select a columns. For examle 0x25 would be the third row (2) and the sixth column (5), giving 0x3f. The standard AES S-box is reported in Figure 16. This S-box hs been obtained by taking, for each byte, its multiplicative inverse in the field. This can bencomputed efficiently via an algorithm that we will see later on, noted b_7,...,b_0, and applying the affine transformation.

$$b_i = b_i \oplus b_{i+4\ mod8} \oplus b_{i+5\ mod8} \oplus b_{i+6\ mod8} \oplus b_{i+7\ mod8} \oplus c_i$$

with c_i representing the i-th bit of 01100011.

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

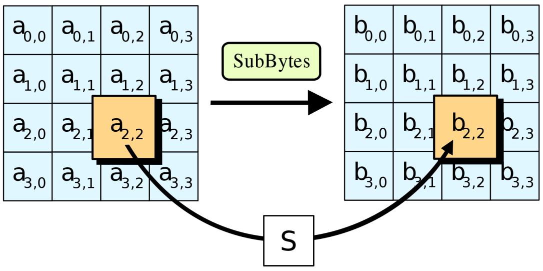
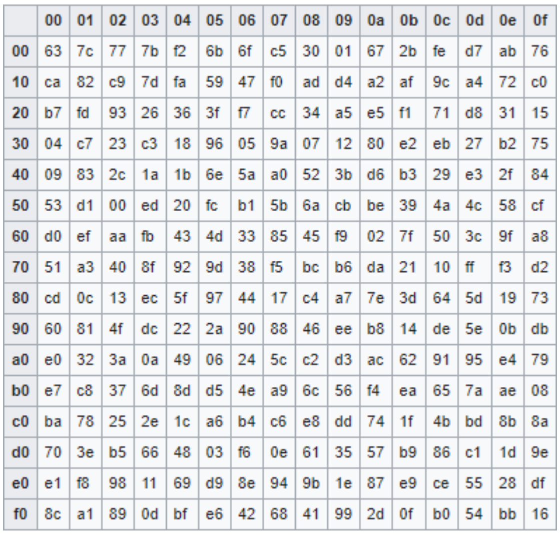


Figure 15: SubBytes.



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

Figure 16: Standard AES S-box.

ShiftRows

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

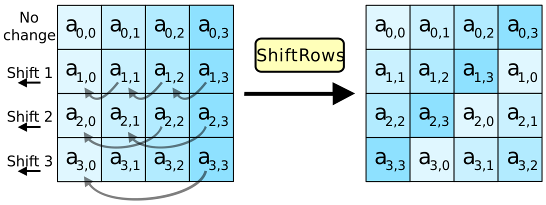

No change
Shift 1
Shift 2
Shift 3
ShiftRows
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,0}$
$a_{2,2}$ $a_{2,3}$ $a_{2,0}$ $a_{2,1}$
$a_{3,3}$ $a_{3,0}$ $a_{3,1}$ $a_{3,2}$

Figure 17: ShiftRows.

MixColumns

Column of the block matrix are multiplied by the following matrix:

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

For example, the first byte of each column is computed as

$$2c_0 \oplus 3c_1 \oplus c_2 \oplus c_3$$

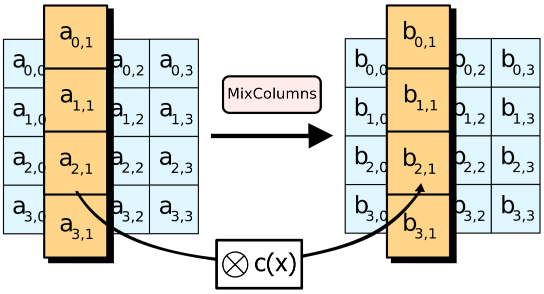

$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
MixColumns
$\otimes$ c(x)
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$

Figure 18: MixColumns.

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 bits key will be 4 words of 4 bytes, i.e., 16 bytes. This is expanded into an array of size 4*(Nr+1), where Nr is the number of rounds. In this way we obtain 4 different words of key for each round.

Here is the overall scheme for AES assuming that variable state is initialized with the 4×4 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])
```

At each round a new key is generated.

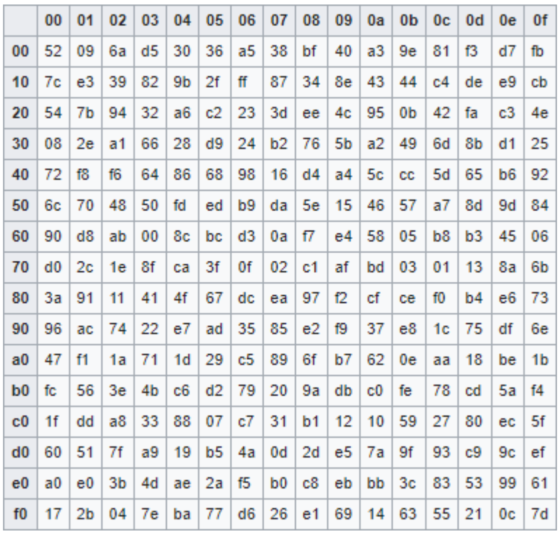
The AES decryption is compute applying inverse operations:

```
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])
```

InvSubBytes is computed by using the inverse substitution of the S-Box in Figure 19.



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

Figure 19: Standard AES inverted S-box.

InvMixColumns is given by the following operation:

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

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 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 InvMixColumns(state roundKey) = InvMixColumns(state) InvMixColumns(roundKey). This allows for inverting the two functions, provided that InvMixColumns is applied to all the round keys.

Call dw the array containing the round keys transformed via InvMixColumns. The final decryption algorithm is:

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

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

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

This is exactly the sam 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.

Exercise 17. Using the algorithm with a,b and p, multiply $(x^4+x^3+1)\times(x^5+1)$

a	**b**	**p**
11001	100001	0
110010	10000	$0\oplus11001=11001$
1100100	1000	11001
11001000	100	11001
110010000	10	11001
(XOR) 10001011	10	11001
100010110	1	11001
(XOR)1101	1	11001
11010		$11001\oplus1101=10100$

In two steps we had a number longer than 8 bits, so we had to xor it with the irreducible polynomial.

Lecture 9

As we said on the previous lecture, the security of AES also depends on the number of rounds: by 2006 the best known attack were on 7 rounds for 128-bits keys, 8 round for 192-bits keys and 9 rounds for 256-bits keys.

Block cipher modes

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. This can be apply to any block cipher (AES, etc.), so encryption and decryption will depend on the chosen one.

Electronic CodeBlock mode (ECB)

This is the simplest mode, it is what we have done so fare with classic ciphers: the plaintext X is split into blocks x_1,..,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.

Example 28. An example is a substitution cipher applied to letters. What we do is to split the plaintext into single letters that are encrypted independently.

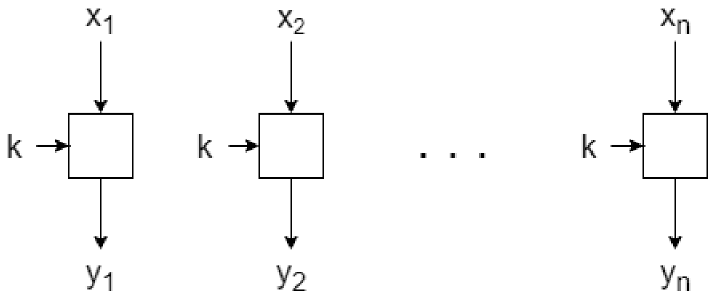


Figure 20: Example of ECB encryption with substitution cipher.

Decryption operation is done by reversing the scheme.

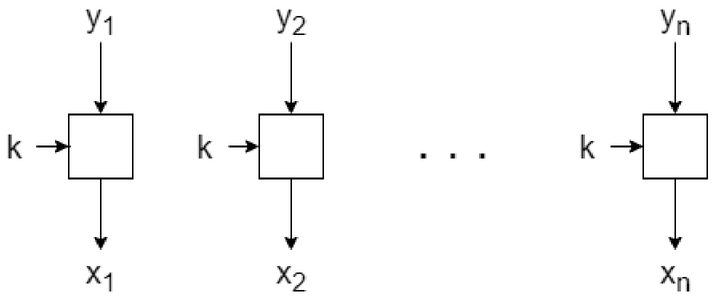


Figure 21: Example of ECB decryption with substitution cipher.

Pros of this approach:

- this scheme has the advantage of being very simple and fast, especially on multi-core computers;
- each single encryption/decryption can be performed independently.

But there are a lot of cons:

- the security of the scheme is poor;
- as monoalphabetic classic ciphers equal plaintext blocks are encrypted in the same way;
- this allows for the construction of a code-book mapping ciphertexts back to plaintexts;
- often in practice part of a plaintext is fixed due (for example) to the message format. E.g., 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;
- 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;
- with information about the format of the plaintext, an attacker might be able to obtain a different meaningful plaintext;
- how critical is this attack depends on the applications, but it is not a good idea to leave such an easy opportunity.

In Figure 22 we can see that large areas with the same uniform colour maintain the same pattern when encrypted using ECB.

The weakness of this method is that we use always the same key.

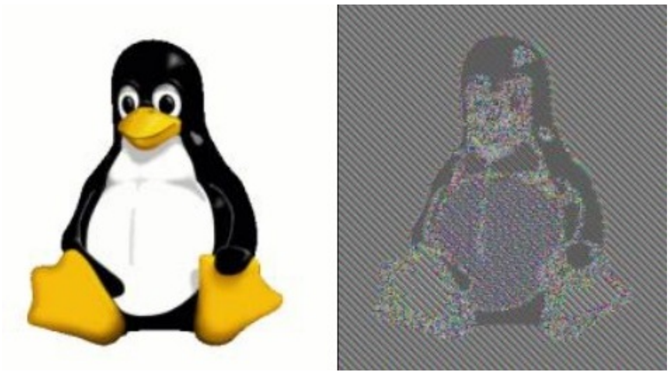


Figure 22: Image encrypted with ECB.

Cipher Block Chaining mode (CBC)

This mode solves or mitigates all the previous issues of ECB, it prevents equal plaintexts to be encrypted the same way and, at the 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 can be sent in clear but has to be random and has to change every time (for every cipher mode).

Encryption and decryption schemas are represented in Figure 23 and in Figure 24.

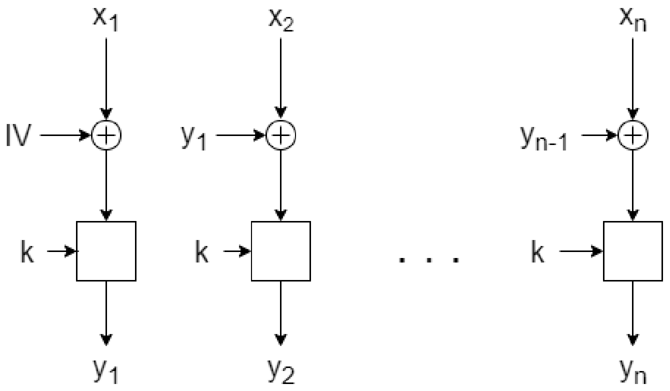


Figure 23: Encryption schema of CBC.

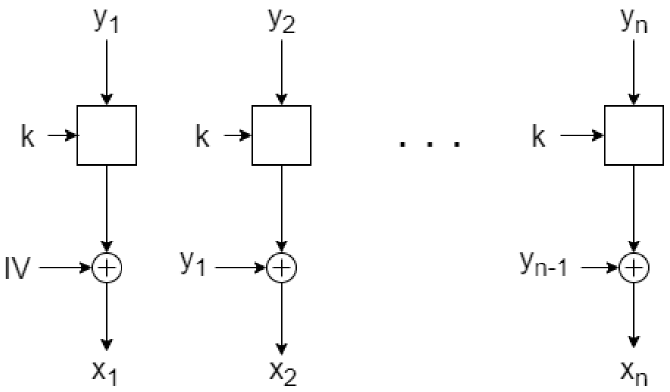


Figure 24: Decryption schema of CBC.

Pros of this method:

- as mentioned above, CBC never encrypts the same plaintext block in the same way, preventing the code-block attack;
- integrity is improved, but is not yet satisfactory. If an attacker swaps, duplicates or eliminates encrypted blocks, this will result in at least one corrupted plaintext block (for example, changing y_1 propagates to y_2). 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 (the application might not check y_1).

The cons are:

- 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 (because we use a xor);
- at the same time x_i is corrupted;
- we cannot do parallel computations.

Exercise 18. Write the expression for CBC encryption and decryption of the i-th block and show, formally, that $\mathrm{D}_k^{CBC}(\mathrm{E}_k^{CBC}(\mathrm{x}_i))=\mathrm{x}_i$. To avoid defining a special expression for y_1, you can let $\mathrm{y}_0=\mathrm{IV}$.

$\mathrm{E}_k^{CBC}(\mathrm{x}_i) = \mathrm{E}_k(\mathrm{x}_i \oplus \mathrm{y}_{i-1})$;
$\mathrm{D}_k^{CBC}(\mathrm{y}_i) = \mathrm{D}_k \oplus \mathrm{y}_{i-1}$.

$$\begin{aligned}
D_k^{CBC}(E_k^{CBC}(x_i)) &= D_k^{CBC}(E_k(x_i \oplus y_{i-1})) && (1)\\
&= D_k(E_k(x_i \oplus y_{i-1})) \oplus y_{i-1} && (2)\\
&= (x_i \oplus y_{i-1}) \oplus y_{i-1} && (3)\\
&= x_i \oplus (y_{i-1} \oplus y_{i-1}) && (4)\\
&= x_i && (5)
\end{aligned}$$

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 XORing 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.

The encryption and decryption schemas of OFB are represented in Figure 25 and in Figure 26.

Notice that the 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.

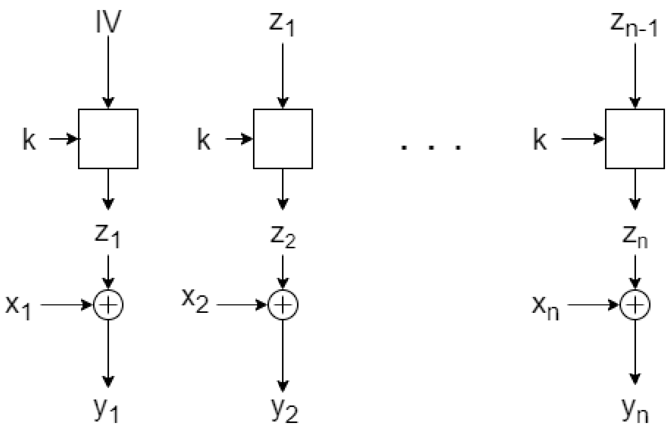


Figure 25: Encryption schema of OFB.

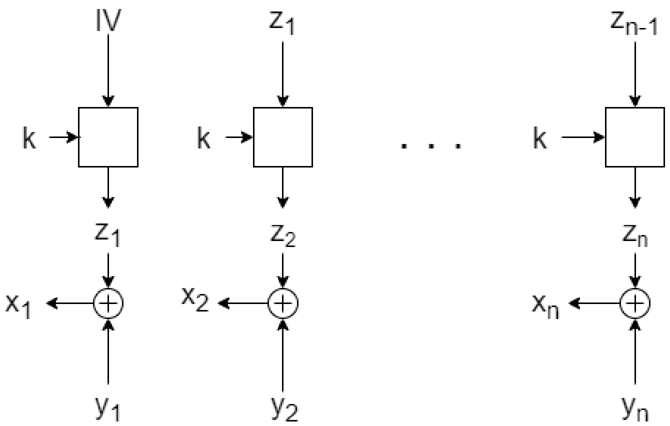


Figure 26: Decryption schema of OFB.

Decryption simply consists of "swapping the arrows" when performing the XOR: ciphertexts are XORed with the key stream to recover the plaintexts.

The generation of the key stream is CBC encryption of a zero plaintext, it is thus possible to reuse CBC implementations to compute it. Notice also that the plaintext blocks can be smaller than 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 a single byte, we have that one key can be slit into 128/8=16 keys of 8 bits, each used to encrypt a single byte.

Pros of this method:

- 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

The cons:

- 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 applications where integrity of the exchanged message is not an issue or is achieved via additional mechanism. 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.

Counter mode (CTR)

It is a variation of OFB where the IV is a random number (nounce) and a counter. The random number can be sent in clear (bit should change at any new stream generation) and the counter changes value during the stream generation. It is widely used in practice.

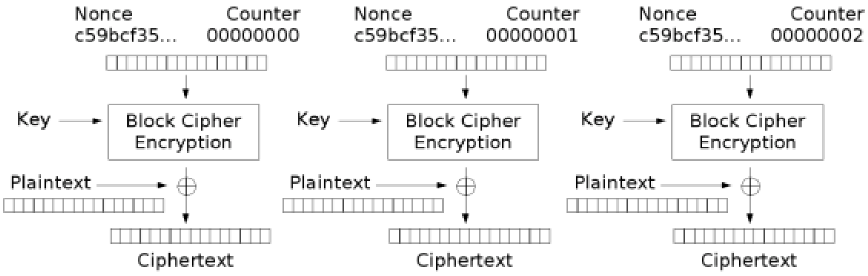


Figure 27: Encryption schema of CTR.

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 (a single byte for example), this mode uses 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 bit of the register.

Encryption and decryption schemas are reported in Figure 28 and in figure 29.

In decryption, as for OFB, the key stream is generated and XORed to the ciphertexts to reconstruct the plaintexts.

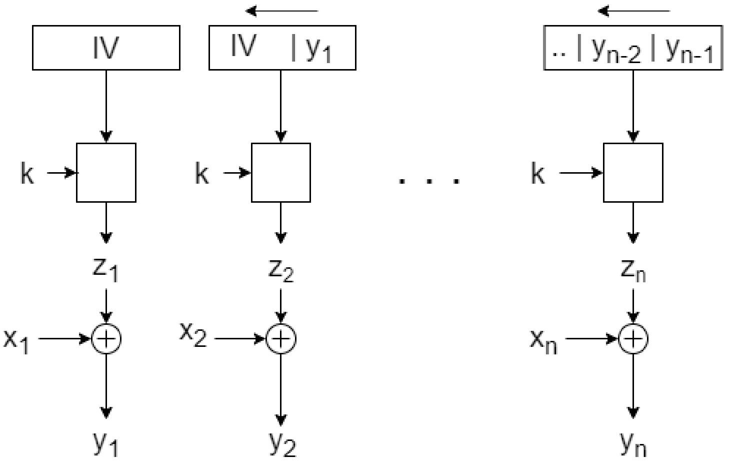


Figure 28: Encryption schema of CFB.

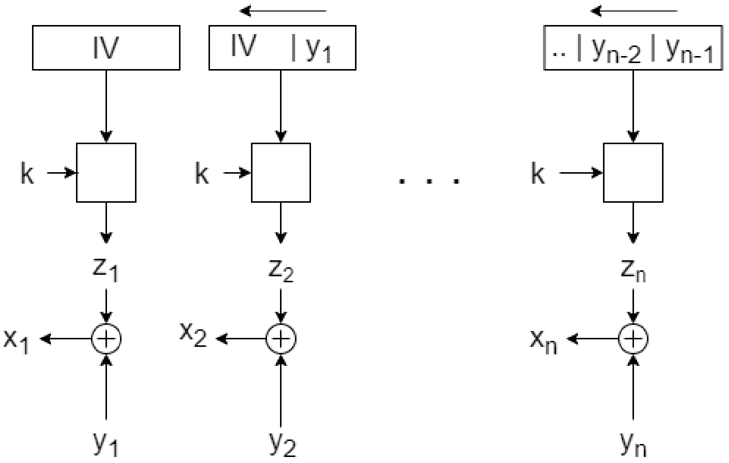


Figure 29: Decryption schema of CFB.

Pros:

- 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 (128 bytes for example) and CSize is the size of the single ciphertext (8 bits for example);
- 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.

Cons:

- 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 transmission (satellite, Tv, ..) it has the problem of propagating an error on a single bit over the next BSize/CSize plaintexts, which are completely corrupted.

Exercise 19. Encode the plaintext "Two One Nine Two" with AES. The key is "Thats my Kung Fu".

First of all we have to transform the plaintext and the key into a Hex notation, using the ASCII table.

Two One Nine Two → `54 77 6F 20 4F 6E 65 20 4E 69 6E 65 20 54 77 6F`
Thats my Kung Fu → `54 68 61 74 73 20 6D 79 20 4B 75 6E 67 20 46 75`

Writing into matrices we have:

$$P = \begin{bmatrix} 54 & 4F & 4E & 20 \\ 77 & 6E & 69 & 54 \\ 6F & 65 & 6E & 77 \\ 20 & 20 & 65 & 6F \end{bmatrix} \qquad K = \begin{bmatrix} 54 & 73 & 20 & 67 \\ 68 & 20 & 4B & 20 \\ 61 & 6D & 75 & 46 \\ 74 & 79 & 6E & 75 \end{bmatrix}$$

Now we have to apply Key expansion for 10 rounds as we are using a 128-bits key.

We AddRoundKey, XORing P and K, to do the XOR, we have to transform Hex to binary and then go back to represent the result:

$$\begin{bmatrix} 00 & 3C & 6E & 47 \\ 1F & 4E & 22 & 74 \\ 0E & 08 & 1B & 31 \\ 54 & 59 & 0B & 1A \end{bmatrix}$$

At this point we apply SubBytes, a fixed non-linear substitution, called S-Box is applied to each byte of the block.

$$\begin{bmatrix} 63 & EB & 9F & A0 \\ C0 & 2F & 93 & 92 \\ AB & 30 & AF & C7 \\ 20 & CB & 2B & A2 \end{bmatrix}$$

ShiftRows: rows of the block matrix are shifted to the left by 0,1,2,3 respectively.

$$\begin{bmatrix} 63 & EB & 9F & A0 \\ 2F & 93 & 92 & C0 \\ AF & C7 & AB & 30 \\ A2 & 20 & CB & 2B \end{bmatrix}$$

MixColumns: columns of the block are multiplied by the following matrix:

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

So:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} 63 & EB & 9F & A0 \\ 2F & 93 & 92 & C0 \\ AF & C7 & AB & 30 \\ A2 & 20 & CB & 2B \end{bmatrix} = \begin{bmatrix} BA & 84 & E8 & 1B \\ 75 & A4 & 8D & 40 \\ F4 & 8D & 06 & 7D \\ 7A & 32 & 0E & 5D \end{bmatrix}$$

At the end we have to XOR again our block with the new key obtained by the Key extension algorithm.

$$\begin{bmatrix} BA & 84 & E8 & 1B \\ 75 & A4 & 8D & 40 \\ F4 & 8D & 06 & 7D \\ 7A & 32 & 0E & 5D \end{bmatrix} \oplus \begin{bmatrix} E2 & 91 & B1 & D6 \\ 32 & 12 & 59 & 79 \\ FC & 91 & E4 & A2 \\ F1 & 88 & E6 & 93 \end{bmatrix} = \begin{bmatrix} 58 & 15 & 59 & CD \\ 47 & B6 & D4 & 39 \\ 08 & 1C & E2 & DF \\ 8B & BA & E8 & CE \end{bmatrix}$$

This is the result of one round, we have to repeat 9 other times to obtain the final matrix.

Lecture 10

In addition to AES there are many other block ciphers in use, such as DES, 3-DES, IDEA, Blowfish and Twofish, RC2, RC5, RC6, ..

Data Encryption Standard (DES)

DES is a symmetric key algorithm, 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 applications, especially in the triple version. DES major problem is the key-length (only 56 bits) that is considered vulnerable with modern parallel computers.

The first attack known in history goes back to January 1999, when distributed.net and the Electronic Frontier Foundation collaborated to publicly break a DES key in 22 hours and 15 minutes. There are also some analytical results which demonstrate theoretical weakness in the cipher, although they are infeasible to mount in practice.

DES 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. The block size is 64 bits an the key length is 56 bits (8 for error correction). It has 16 identical rounds, the encryption starts with an initial permutation (IP) and ends with a final permutation (FP, inverse operation). Before the main rounds, the block is divided into two 32-bit halves and processed alternatively. This criss-crossing is known as the **Feistel scheme**.

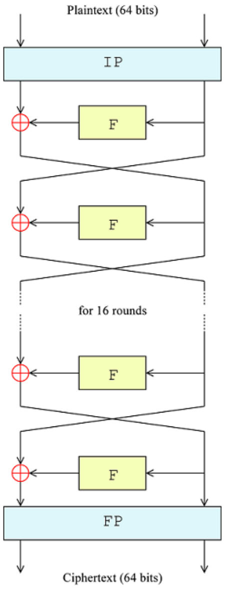


Figure 30: DES encryption schema.

The Feistel function (F)

Feistel function consists of different operations: Expansion (from 32 to 48 bits), Key-mixing (key schedule), Substitution S-box (results 32 bits) and Permutation. The schema of the function is represented in Figure 31.

The alternation of substitution from the S-boxes, and permutation of bits from the P-box and E-expansion provides the so-called **"confusion and diffusion"** (Shannon).

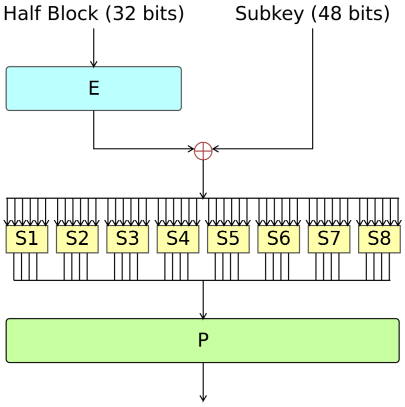


Figure 31: Feistel function.

Confusion is the process that drastically changes data from 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.

International Data Encryption Algorithm (IDEA)

IDEA was proposed in 1990 as a substitute of DES and it is currently adopted in many applications. It is not based on non-linear substitutions (S-Boxes), confusion and diffusion are obtained by a combination of three operations: xor, sum and multiplication modulo 2^{16}. Patent issues have reduced the popularity of this cipher. Compared to other, IDEA performance is not so high.

Blowfish and Twofish

Blowfish has been proposed in 1993, 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. This 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 "successor" of Blowfish, both ciphers have been developed by Bruce Schneier.

RC2, RC5, RC6

They are a family of ciphers developed by Ron Rivest.

- RC2 (1987): it is vulnerable to a related-key attack using 2^{34} chosen plaintext;
- RC5 (1994): it uses data dependent rotation, it is extremely simple but it 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 (derived from RC5): it has been one of the AES finalists.

3DES

3DES is a triple iteration of DES, it is implemented, for example, in SSH, TLS/SSL and it is adopted in many commercial applications. Bank circuit and credit card issuers use 3DES in smartcard-based applications and for PIN protection.

The aim of 3DES is to increase 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-bits key could be broken in a 2^{56} time/space complexity brute force attack.

One technique is to strengthen ciphers in iteration. All modern ciphers are based on rounds, i.e., repetitions of the same core algorithm. What happens if we iterate a while cipher such as DES or AES? It increases the key length. DES has 56-bit key that is considered weak nowadays. If we iterate the cipher three times using different keys, we obtain (k_1, k_2, k_3) of 168 bits which is too hard to break.

We know that iteration make sense only if the cipher is not idempotent. The following informal argument suggests that modern ciphers are very unlikely to be idempotent. We reason on DES bu the same reasoning would apply to different block ciphers.

DES is non-idempotent

DES has a block size of 64 bits, if we list all the 2^{64} possible blocks and we pick one DES key k, 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:

$$\begin{gathered} 0 \rightarrow \rho(0) \\ 1 \rightarrow \rho(1) \\ 2 \rightarrow \rho(2) \\ \dots \\ 2^{64}\text{-}1 \rightarrow \rho(2^{64}-1) \end{gathered}$$

The number of permutations of 2^{64} elements is $2^{64}!$ which is enormously big compared to the 2^{56} DEs keys. 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.

The probability that the composition of two such permutations is still in this subset (i.e., DES is idempotent) is $2^{56}/2^{64}!$, which is a 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. Thus DES is non-idempotent (there exists also formal proofs).

Meet-in-the-middle

Meet-in-the-middle 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 to select one pair, say (X,Y), and try to decrypt Y with all the possible second keys k_2. All the resulting values Z are stored into a table together with the key, which is indexed by Z.

Z	key
$D_0(Y)$	0
$D_1(Y)$	1
...	...
$D_{2^{56}-1}(Y)$	2^{56}-1

Now we try to encrypt under all the possible first keys k_1 the plaintext X and we look the obtained value into the table. If we find a match, we test the resulting pair (k_1,k_2) on all the other plaintext/ciphertext pairs and, if all the tests succeeds, we give it as output.

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 has 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 (vs brute force).

False keys

It is very important that, whenever a pair (k_1,k_2) 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. i.e., 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 these assumptions, 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 $1/2^{64}$. 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.

Lesson learned

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 (k_1,k_2,k_3).

The meet-in-the-middle attack is still possible in 3DES but it reduces the cost in time to 2^{112} with a table of size 2^{56} entries. The idea is to build the table by decrypting Y under all k_3 and then try all the pairs (k_1,k_2).

Exercise 20. Encode message M=(3,2) using the Hill cipher with key matrix

$$K = \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}$$

Decode the found encrypted message.

$$E_k(3,2) = (3,2) \times \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix} \text{mod} 26 = (9+2, 3+4)\text{mod}26 = (11,7)$$

Decryption:

$$K^{-1} = \det^{-1}(K) \begin{bmatrix} 2 & -1 \\ -1 & 3 \end{bmatrix} \text{mod}26 = \det^{-1}(K) \begin{bmatrix} 2 & 25 \\ 25 & 3 \end{bmatrix} \text{mod}26$$

$$\det(K) = (6-1)\text{mod}26 = 5$$

$$\det^{-1}(K) = 21, (21 \times 5)\text{mod}26 = 1$$

$$K^{-1} = 21 \begin{bmatrix} 2 & 25 \\ 25 & 3 \end{bmatrix} \text{mod}26 = \begin{bmatrix} 42 & 525 \\ 525 & 63 \end{bmatrix} \text{mod}26 = \begin{bmatrix} 16 & 5 \\ 5 & 11 \end{bmatrix}$$

$$D_k(11,7) = (11,7) \begin{bmatrix} 16 & 5 \\ 5 & 11 \end{bmatrix} \text{mod}26 = (176{+}35, 55{+}77)\text{mod}26 = (211,132)\text{mod}26 = (3,2)$$

Lecture 11

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.

It 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, for N users, this would require the exchange of N(N-1)/2 keys, $O(N^2)$. From another point of view, this is a complete graph.

Example 29. If we have N=4, we would have $\frac{4\times 3}{2} = 6$ channels and 6 pairs of symmetric keys.

Symmetric key ciphers make sense until we have a reasonable number of users.

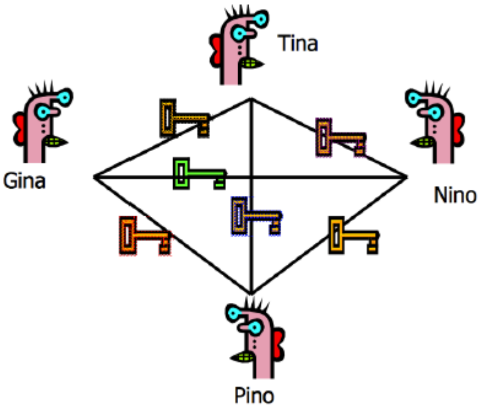


Figure 32: Symmetric key graph.

For example in a LAN with 1000 users, we would have ≈ 500000 keys. these keys should be pre-distributed to users in a secure way (e.g., offline) but this is totally impractical and would never scale on a wide-area network such as the Internet.

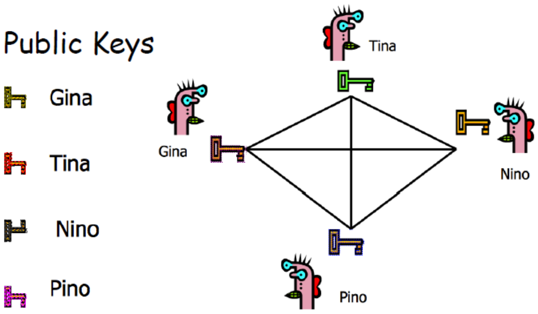


Figure 33: Public keys idea.

To overcome this problem, in 1976, Whitfield Diffie and Martin Hellman proposed the asymmetric-key ciphers. The idea is the following: a user A has one encrypting and one decrypting key. They are different but correlated (this is why they are called asymmetric). The encrypting key is public, the decrypting key is a secret, known only by A.

The public key is published in a public list and is known by everybody, even the attacker. Public and private keys are correlated but the knowledge of the public key does not give any information about the private key.

A schema of public key cryptography is shown in Figure 34.

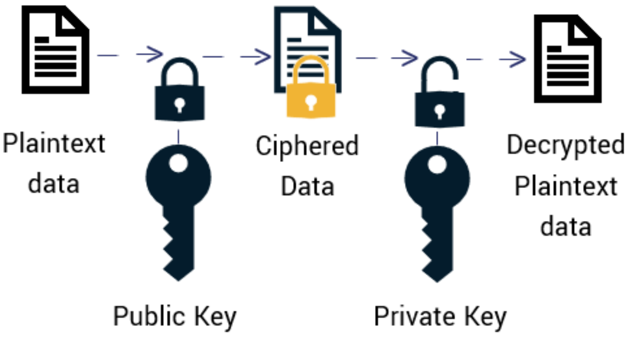


Figure 34: Public key encryption schema.

We define the public key of A with $\mathbf{PK}_A$ and the secret key of A with $\mathbf{SK}_A$. If B wants to send a message to A, he sends $E_{PK_A}(M)$, when A receives the messages, he decrypt it as $D_{SK_A}(E_{PK_A}(M))$.

Encryption and decryption algorithms are defined such that $D_{SK_A}(E_{PK_A}(M)) = M$ holds.

Definition 16. An **asymmetric-key cipher** is a quintuple $(P,C,K_S \times K_P,E,D)$ with $E:K_P \times P \rightarrow C$ and $D:K_S \times C \rightarrow P$ and such that:

- it is computationally easy to generate a key-pair (SK,PK) in $K_S \times K_P$;
- it is computationally easy to compute $y=E_{PK}(x)$;
- is is computationally easy to compute $x=D_{SK}(y)$.

As said before, $D_{SK}(E_{PK}(x))=x$. It is computationally infeasible to to compute SK knowing PK and y. It is also infeasible to compute $D_{SK}(y)$ knowing PK and y and without knowing SK.

As encryption is performed under PK while decryption under SK, the decryption key is now different from the encryption key.

What security properties do we have? We achieve secrecy because the attacker is not able to decrypt the message as it doesn't have the secret key SK. On the other hand we cannot assure authentication, everybody can send a message to A saying it is B (with symmetric keys this does not happen). For this reason we need a digital signature.

One-way-trap-door functions

Definition 17. An injective, invertible function is **one-way** iff:

- $y=f(x)$ is easy to compute;
- $x=f^{-1}(y)$ is infeasible to compute.

Definition 18. An injective, invertible family of functions f_k is **one-way trap-door**, iff given k:

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

The trap-door is a hidden way to go back to the preimage x of the function, only knowing the trap-door we can compute $f_k^{-1}(y)$.

The Merkle-Hellman knapsack system

This cipher has been broken, but still gives an idea of how asymmetric-key ciphers relate to one-way-trap-door functions. It 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 $\sum_{i=1}^{n} x_i s_i = T$.

Example 30. 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.
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 to invert the function? The trick is to start from a specific instance of the problem that it 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. This means that 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. The code for the algorithm is the following:

```
def subsetSum(S,T):     #assumes S is a super-increasing list of integers
  x=[]
  S.reverse()           #reverse the 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             #subtract 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
```

The cipher

- We start from a super-increasing problem $(s_1,..,s_n)$;
- we choose a prime $\mathbf{p}>\sum_{i=1}^{n} s_i$;

- we choose a random **a** such that 1<a<p;
- we transform the initial super-increasing problem into $\hat{s_1}, .., \hat{s_n}$, with $\hat{s_i}$=as$_i$ mod$p$. Notice that this problem is not super-increasing in general.

Example 31. Let (1,2,5,12) be a super-increasing sequence, p=23 and a=6. Compute s$_i$a mod$p$.

$$(1*6) \text{ mod } 23 = 6$$
$$(2*6) \text{ mod } 23 = 12$$
$$(5*6) \text{ mod } 23 = 7$$
$$(12*6) \text{ mod } 23 = 3$$

Thus we obtain the sequence (6,12,7,3).

The trap-door is composed of (s$_1$,..,s$_n$), p and a, that are kept secret.
The public key is ($\hat{s_1}, .., \hat{s_n}$).

To encrypt we first have to translate the message from ASCII to binary obtaining x$_1$,..x$_n$ and then we can encrypt it as follows:

$$E_{PK}(x_1,..,x_n) = \sum_{i=1}^{n} x_i\hat{s_i}$$

For example, given the plaintext (1,0,0,1) and the sequence (6,12,7,3) we have E$_{PK}$(1,0,0,1) = 1*6 + 0*12 + 0*7 + 1*3 = 6 + 3 = 9.

The decryption works as follows: D$_{SK}$(y) is the solution of the super-increasing problem (s$_1$,..,s$_n$) with target a^{-1}y mod$p$.

Example 32. a^{-1} is 4, since a=6, p=23, 6*4 mod 23 = 1.

Given E$_{PK}$(1,0,0,1) = 9, to decrypt we have to compute the target a^{-1}*9 mod$p$ = 4*9 mod 23 = 13.

We finally solve the super-increasing problem (1,2,5,12) with target 13 with target 13: subsetSum([1,2,5,12],13) = [1,0,0,1] which gives the initial plaintext (1,0,0,1).

Exercise 21. 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$_i$*a mod$p$.
Then compute the encryption and the decryption of letter "a".

Public key:

$$2*588 \text{ mod } 881 = 295$$
$$7*588 \text{ mod } 881 = 592$$
$$11*588 \text{ mod } 881 = 301$$
$$21*588 \text{ mod } 881 = 14$$
$$42*588 \text{ mod } 881 = 28$$
$$89*588 \text{ mod } 881 = 353$$
$$180*588 \text{ mod } 881 = 120$$
$$354*588 \text{ mod } 881 = 236$$

So the public key is (295,592,301,14,28,353,120,236).

Letter "a" in binary is (0,1,1,0,0,0,0,1) Encrypting it we obtain:
$E_{PK}(0,1,1,0,0,0,0,1) = 592 + 301 + 236 = 1129$.

To decrypt, we need to find a^{-1}, it is 442, since 588*442 mod 881 = 1. The target is: 442*1129 mod 881 = 372. subsetSum([2,7,11,21,42,89,180,354],372) = [0,1,1,0,0,0,0,1] and this is the binary value of letter "a".

Lecture 12

In the previous lecture we said that to decrypt we use a^{-1}y mod$p$. But why?

Proof. The correctness of the above cipher can be proved as follows:

$$\begin{aligned}
E_{PK}(x_1,..,x_n) &= \sum_{i=1}^{n} x_i \hat{s}_i && (1)\\
&= \sum_{i=1}^{n} a x_i s_i \text{ mod} p && (2)\\
&= a \sum_{i=1}^{n} x_i s_i \text{ mod} p && (3)\\
a^{-1} E_{PK}(x_1,..,x_n) \text{ mod} p &= a^{-1} a \sum_{i=1}^{n} x_i s_i \text{ mod} p && (4)\\
&= \sum_{i=1}^{n} x_i s_i \text{ mod} p && (5)\\
&= \sum_{i=1}^{n} x_i s_i && (6)
\end{aligned}$$

We can ignore mod$p$ since p>$\sum_{i=1}^{n}$s$_i$, meaning that the transformed target a^{-1}y mod$p$ is, in fact, the target for the initial, easy problem.

■

RSA cipher

RSA stands for Rivest-Shamir-Alderman, the authors of the cipher in 1978. RSA is a public key cipher that allows to encrypt messages using an algorithm that uses some prime numbers, it is still unviolated.

The Euler function

The Euler function ϕ(n) return the number of numbers ≤n that are coprime to n.

Recall: **i** and **n** are coprime iff gcd(i,n)=1, i.e., if the only common divisor is 1.

Example 33. ϕ(3)=2 since 1 and 2 are coprime to 3, ϕ(4)=2 since only 1 and 3 are coprime to 4 and so on.

Some values of ϕ(n) are reported in the following table.

n	**ϕ(n)**
1	1
2	1
3	2
4	2
5	4
6	2
7	6

Notice that, if n is prime then ϕ(n)=n-1. In fact, by definition, a prime number is coprime to all the numbers smaller than n.

Moreover when n = p_1 * ... * p_k with $p_1 \neq p_2 \neq ... \neq p_k$ and p_i primes, we have that ϕ(n) = ϕ(p_1) * ... * ϕ(p_k) = (p_1-1) * ... * (p_k-1).

Example 34. n=3*15=15, ϕ(15)=2*4=8?
The numbers $\leq$15 that are coprime to 15 are: 1,2,4,7,8,11,13,14.

Proof. Proof for n=pq with p$\neq$q prime: the numbers $<$ n that are not coprime to n are the multiples of p and q, i.e., p, 2p, ... (q-1)p, q, 2q, ..., (p-1)q that are (q-1)(p-1).

Now we have that ϕ(n) = pq-1-(q-1)-(p-1) = pq-1-q-1-p+1 = (p-1)(q-1).

■

Key generation

- We generate two distinct and big prime numbers p and q;
- we compute n=pq and ϕ(n)=(p-1)(q-1);
- we choose a small **a**, prime with ϕ(n) and smaller than ϕ(n);
- we compute the unique **b** such that a*b modϕ(n)=1.

At this point we can say that PK=(b,n), SK=(a,n); we assume C=P=Z_n.

- Encryption of x: E_{PK}(x) = x^b mod$n$;
- Decryption of y: D_{SK}(y) = y^a mod$n$.

Example 35. We generate two distinct numbers, p and q

For example p=5 and q=11 (small numbers in this example).

We compute n=pq=55 and ϕ(n)=(p-1)(q-1), ϕ(55)=4*10=40.

We choose a small a, prime with ϕ(55) and $<$ ϕ(55)=40, for example a=23.

We compute the unique b such that 23*b modϕ(55)=1, e.g., b=7 as 23*7 mod 40 = 161 mod 40 =1.

PK=(b,n)=(7,55); SK=(a,n)=(23,55). We assume C=P=Z_{55}.

Encryption of x: E$_{PK}$(x) = x^7 mod 55
For example if x=2, E$_{PK}$(2) = 2^7 mod 55 = 128 mod 55 = 18.

Decryption of x: D$_S K$(y) = 18^{2}3 mod 55 = 2.

Example 36. Generate two distinct prime numbers, p=5 and q=7. Compute possible PK, SK, E$_{PK}$(x), D$_{SK}$(y) with x=2.

We compute n=pq=35 and ϕ(n)=(p-1)(q-1)=24.

We choose a small a prime with ϕ(n) and $<$ ϕ(n), for example a=11

We compute the unique b=11 as 11*11 mod 24 = 1.

PK=(11,35), SK(11,35)

E$_{PK}$(2) = 2^{1}1 mod 35 = 18.
D$_{SK}$(18) = 18^{1}1 mod 35 = 2.

To make the cipher secure, RSA requires a big modulus n of at least 1024 bits. However, with these sizes, implementation becomes an issue. E.g., a linear complexity O(n) is prohibitive as it could require 2^{1024} steps. Every operation should in fact be polynomial with respect to the bit-size k.

Basic operations such as sum, multiplication and division can be performed in O(k), O(k^2), O(k^2) steps respectively, by using simple standard algorithms (the one we use when we compute operations by hand). Reduction modulo amounts to compute a division which is, again, O(k^2).

Exponentiation 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 multiplications would then require k^2*2^k operations, 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 ways to compute this operation.

Square-and-Multiply algorithm

The idea is: 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=x*x*x*x*x*x*x*x which is just 3 multiplications.

If the exponent is not a power of 2, we can exploit a similar trick by performing some additional multiplications when needed.

E.g., x^{10} can be done 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 by 2 and we square, if it is odd, we get one out (which is the 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.

Exercise 22. Try to compute 2^{10} where 10 is 1010, x=2.

r	bit	r
1*1=1	1	1*2=2
2*2=4	0	
4*4=16	1	16*2=32
32*32=1024	0	

The Python code of the algorithm is the following:

```
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
```

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.

Lecture 13

Exercise 23. Perform encryption and decryption using the RSA algorithm for the following:

$$p=7,\ q=11,\ a=7,\ x=2$$

We compute n=pq=77 and ϕ(n)=(p-1)(q-1)=60.

We compute the unique b=43 as 7*43 mod 60 = 1.

PK=(43,77), SK(7,77)

E$_{PK}$(2) = 2^43 mod 77 = 30.
D$_{SK}$(30) = 30^7 mod 77 = 2.

Exercise 24. Try to compute 2^8 where 8 is 1000 and x=2.

We can use the squareAndMultiply algorithm:

r	**bit**	**r**
1*1=1	1	1*2=2
2*2=4	0	
4*4=16	0	
16*16=256	0	

The result is thus 2^8=256.

Correctness of RSA cipher

We want to show that decrypting under the private key a plaintext x encrypted under the public key gives x.

Theorem 3 (The Euler Theorem). Let a and n be coprime, i.e., gcd(a,n)=1. Then $a^{\phi(n)} \mathrm{mod} n=1$.

Proof. Let S=(s$_1$, ..., s$_{\phi(n)}$) be the ϕ(n) numbers less than n and coprime with n. We consider R=(a*s$_1$mod$n$, ..., a*s$_{\phi(n)}$mod$n$) and we show that S=R. We need a few lemmas.

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

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

■

Example 37. Let's consider x=7, y=3, p=5, q=11 and n=55.

x=7 and y=3 are coprime to 55, what about x*y=21? gcd(21,55) = gcd(7*3,5*11).

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

Proof. Since x mod$n$=x-k*n, we have that any common divisor d of x mod$n$ and n must divide x. In fact $\frac{x \text{ mod} n}{d} = t = \frac{x-kn}{d} = \frac{x}{d} - kr$, 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.

■

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

Proof. We have a*x-k*n=a*x mod$n$=a*y mod$n$=a*y-j*n. Thus a*x-a*y=w*n (w=k-j) that implies a(x-y)/a=w*n/a and so x-y=w*n/a. Since gcd(a,n)=1, we have that a must divide w and so x-y=t*n, i.e., (x-y)mod$n$=t*n mod$n$=0, i.e., x mod$n$=y mod$n$.

■

Exercise 25. Given a=3, n=6 find x and y (different) such that a*x mod$n$=a*y mod$n$.

For example x=1 and y=3.
In this case the Lemma 3 doesn't hold because gcd(a,n)=gcd(3,6)=3≠1.

Exercise 26. Find a coprime to n (n=6), and find x and y such that a*x mod$n$=a*y mod$n$.

We could choose a=5, it is coprime to 6. Values for x and y are x=1 and y=7.
That is x mod$n$ = 1 mod5 = 1 = y mod$n$ = 7 mod 6.

Putting all together:
by Lemma 1 (Let x,y be coprime to n, then x*y is coprime to n) and 2 (Let x be coprime to n, then x mod$n$ is coprime to n) we have that all numbers in set R = (a*$s_1$mod$n$, ..., a*$s_{\phi(n)}$mod$n$) 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⊆S.
Now, consider a*s_i mod$n$ and a*s_j mod$n$ in R. Since a is coprime to n (by hypothesis), by Lemma 3 we have that a*s_i mod$n$=a*s_j mod$n$ implies s_i mod$n$=s_j mod$n$, which gives a contradiction. Thus we have that a*s_i mod$n$≠a*s_j mod$n$ for all i and j. This proves that R=S.

To conclude the proof, it is enough to observe that since R=S:

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

Since all s_i are coprime to n, by applying many times Lemma 3 (we divide both sides by s_1, then s_2, ...), we obtain 1=$a^{\phi(n)}$ mod$n$, which is what we wanted to prove.

■

Example 38. Let p=2, q=3. what happens if a=3 not coprime to n=2*3=6?
$a^{\phi(n)}$ mod$n$ = $3^{\phi(n)}$ mod6 = 3^2 mod6 = 3 $\neq$ 1.

Let's consider a=5 (5 is coprime to 6):
$a^{\phi(n)}$ mod$n$ = $5^{\phi(n)}$ mod6 = 5^2 mod6 = 1.

Lecture 14

Exercise 27. Perform the encryption and decryption using the RSA algorithm for the following:

- p=3, q=11, a=7, x=5

  We compute n=pq=33 and ϕ(n)=(p-1)(q-1)=20.

  We compute the unique b=3 as 7*3 mod 20 = 1.

  PK=(3,33), SK(7,33)

  $E_{PK}(5) = 5^3$ mod 33 = 26.
  $D_{SK}(26) = 26^7$ mod 33 = 5.

- p=5, q=11, b=3, x=9

  We compute n=pq=55 and ϕ(n)=(p-1)(q-1)=40.

  We compute the unique a=27 as 27*3 mod 40 = 1.

  PK=(3,55), SK(27,55)

  $E_{PK}(9) = 9^3$ mod 55 = 14.
  $D_{SK}(14) = 14^27$ mod 55 = 9.

Exercise 28. Try to compute x^{15} where 15 is 1111.

We can use the squareAndMultiply algorithm:

r	**bit**	**r**
1*1=1	1	1*x=x
$x*x=x^2$	1	$x^2*x=x^3$
$x^3*x^3=x^6$	1	$x^6*x=x^7$
$x^7*x^7=x^{14}$	1	$x^{14}*x=x^{15}$

Exercise 29. Assume that in our RSA public key cryptosystem the public key is (b,n)=(141,391).

- Find the deciphering key (a,n).
- Assume that you have received the enciphered message y=306. Decipher it.
- Why is this encryption not secure?

Since n=391 can only be obtained from 17*23, we have now the values of p and q.
At this point we can compute ϕ(n)=(p-1)(q-1)=352.
We then compute the unique a=5 as 5*141 mod 352 = 1.

At this point we have the secret key (a,n)=(5,391). We can decrypt 306 this way:
$D_{SK}(306)=306^5 \bmod 391 = 17$.

This encryption is not secure because n is too small and an attacker can simply factorize it to obtain p and q.

Exercise 30. Consider a composition of this substitution cipher with this key:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
                         ↓
Q P E O X C D N J M W H S A R Y T F B K I L G Z U V
```

with a shift cipher with k=7.

- Show the encryption of the plaintext `JAVA` using the composition of the two ciphers.

  $$\begin{aligned} J &\rightarrow M + 7 = T \\ A &\rightarrow Q + 7 = X \\ V &\rightarrow L + 7 = S \\ A &\rightarrow Q + 7 = X \end{aligned}$$

- Show that you can obtain the same encryption with a unique substitution cipher and show the new key.

  We can shift the substitution cipher's key by 7 obtaining the following new key:

  ```
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
                           ↓
  X W L V E J K U Q T D O Z H Y F A M I R P S N G B C
  ```

  If we encrypt JAVA with this new key we obtain:

  $$\begin{aligned} J &\rightarrow T \\ A &\rightarrow X \\ V &\rightarrow S \\ A &\rightarrow X \end{aligned}$$

Show the encryption of the plaintext `JAVA` using the composition of the two ciphers.

Exercise 31. Consider a cipher in which plaintexts and ciphertexts are of **d** bits (|P|=|C|=2^d), and the keys are of e≤d bits (|K|=2^e). The encryption function is:
$E_{k_0,..,k_{e-1}}(x_0,...,x_{d-1}) = x_0 \oplus k_{0 \bmod e},..,x_i \oplus k_{i \bmod e},...,x_{d-1} \oplus k_{(d-1) \bmod e}$.

- Consider the case d=8 and e=1 and show the two possible encryptions of plaintext 11001011.

  With e=1 we have 2^1=2 different keys of length 1, assuming that k_0=0 and k_1=1 we obtain the following encryption:
  $E_{0,1}(11001011) = 11001011$

  On the other hand, if we consider k_0=1 and k_1=0 we will obtain:
  $E_{1,0}(11001011) = 00110100$.

- Show with a counterexample that if e<d the cipher is not perfect.

  We remember that a cipher is perfect iff Pp(x|y)=P$_p$(x) for all x in P and i in C.

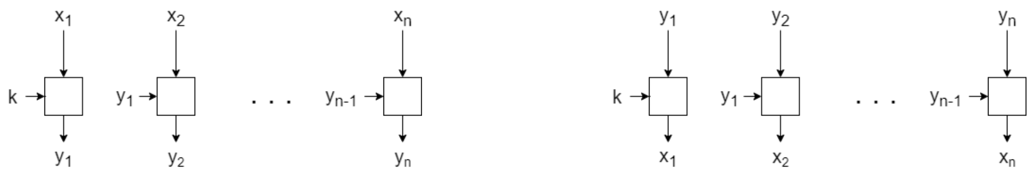
  We can take e=d-1. In this case (d-1) mod e = 0, thus k$_0$=k$_{d-1}$

  Let's assume as a possible plaintext 11010100 with d=8 and P$_p$(11010100)>0. Then P$_p$(x|y)=P$_p$(11010100|00000000)=0 (this can never happen). For this reason it is not a perfect cipher.

Exercise 32. It is given an encryption scheme called CKC (cipher key changing) defined as follows. Given a X=x$_1$,...,x$_n$, we compute Y=y$_1$,...,y$_n$ as:

- y$_1$ = E$_k$(x$_1$);
- y$_i$ = E$_{y_{i-1}}$(x$_i$) for all 1<i≤n.

Draw the encryption and decryption schema of CKC and discuss its (in)security.



(a) Encryption.

(b) Decryption.

Figure 35: Encryption and decryption schema of CKC.

This cipher is insecure because with y$_i$ an attacker can decrypt y$_i+1$ and all the following blocks.

Exercise 33. It is proposed an electronic voting scheme where the votes of the electors are sent to a server with a simple encryption E$_k$(v), where v is the id of a candidate. Assume an attacker can intercept messages to the server, show a simple attack scenario (similar to the codebook attack) where the secrecy of the votes is violated without finding k or cryptoanalyzing the cipher. Propose a technique that improves the security of the protocol.

The number of possible candidates is limited. An attacker can send his/her vote x, see E$_k$(x) and find how many people have voted in the same way (violates the privacy). If the possible candidates are two he can know the final result in advance.

To improve the security of the protocol, we can use a stream cipher with random IV, add random bytes to the name of the candidate, etc..

Lecture 16

Lecture 15 is not present, as it was an exercises class in preparation for the midterm.

Proven the Euler Theorem (3), we now want to prove that, using RSA, if we decrypt and encrypted message, we go back to the initial message.

We have that $\mathrm{D}_{SK}(\mathrm{E}_{PK}(\mathrm{x}))$ = $(\mathrm{x}^b \bmod \mathrm{n})^a \bmod \mathrm{n}$ = $(\mathrm{x}^b)^a \bmod \mathrm{n}$ = $\mathrm{x}^{ab} \bmod \mathrm{n}$. We want to prove that $\mathrm{x}^{ab} \bmod \mathrm{n}$ = x, that implies $\mathrm{D}_{SK}(\mathrm{E}_{PK}(\mathrm{x}))$ = x. To prove this, we will use the Euler Theorem. Note that ab mod ϕ(n)=1 implies ab=kϕ(n)+1 for some k.

Correctness of RSA. First of all we can notice that

$$x^{ab} \bmod n = x^{k\phi(n)+1} \bmod n = (x^{\phi(n)})^k x \bmod n.$$

As we have to prove that $(x^{\phi(n)})^k x \bmod n = x$ and x is different from 0, we can simplify it to $(x^{\phi(n)})^k \bmod n = 1$. We split the proof in two cases:

- **gcd(x,n)=1.** In this case we know that the Euler theorem holds, and we directly have that $\mathrm{x}^{\phi(n)} \bmod \mathrm{n}$ = 1, which implies $(\mathrm{x}^{\phi(n)})^k \bmod \mathrm{n}$ = 1, giving the thesis;
- **gcd(x,n)>1.** Since x<n, and we know that n=pq, we have that either gcd(x,n)=p or gcd(x,n)=q. We assume gcd(x,n)=p (the proof for the other case is identical).

  We can write x=pj for some j. We have that gcd(x,q)=1 (as x<n) that, for Euler Theorem, gives $\mathrm{x}^{\phi(q)} \bmod \mathrm{q}$ = 1.

  This implies $\mathrm{x}^{\phi(q)\phi(p)} \bmod \mathrm{q}$ = $\mathrm{x}^{\phi(n)} \bmod \mathrm{q}$ = 1 (since p and q are primes, $\mathrm{x}^{\phi(n)}$ = $\mathrm{x}^{\phi(q)\phi(p)}$, moreover $1^{\phi(p)} \bmod \mathrm{q}$ = 1).

  This implies $(\mathrm{x}^{\phi(n)})^k \bmod \mathrm{q}$ = 1, giving $(\mathrm{x}^{\phi(n)})^k$ = wq + 1 for some w, by definition of modulus.

  At this point we obtain:

$$\begin{aligned}(x^{\phi(n)})^k\ x \bmod n &= (wq+1)x \bmod n\\ &= (wq+1)jp \bmod n\\ &= (wqjp+jp) \bmod n\\ &= (wqjp+x) \bmod n\\ &= (njw+x) \bmod n\\ &= x.\end{aligned}$$

Thus $\mathrm{D}_{SK}(\mathrm{E}_{PK}(\mathrm{x})) = \mathrm{x}$ and the RSA cipher is correct. ■

Example 39. Let's see an example with p=3, q=11, x=2. PK = (b,n) = (3,33); SK = (a,n) = (7,33).
ϕ(n)=(p-1)(q-1)=2*10=20
x^{ab} mod n = $\mathrm{x}^7 * 3$ mod 33 = x^{21} mod 33 = 2^{21} mod 33 = 2097152 mod 33 = 2.

Inverse modulo

We have seen that sometimes we need to compute the inverse of a number, for example in RSA we need to compute *a* and *b*, that are one the inverse of the other in module ϕ(n). Usually we choose *b* and then we compute its inverse to recover *a*. When gcd(b,ϕ(n))=1, the inverse is guaranteed to exist by the Euler Theorem. In this particular case there is also an efficient algorithm to compute the inverse based on the Euclidean algorithm for computing gcd.

Euclidean algorithm

The idea is to compute the gcd(c,d) (when d$\neq$0) as gcd(d, c mod d), since it can be easily seen that they are the same.

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

Example 40. Let's try to compute gcd(15,5):

- `Euclid(15,5)`
- d=5$\neq$0
- tmp=c%d=15%5=0
- c=d=5
- d=tmp=0
- return 5

Thus gcd(15,5)=5.

This algorithm has a very good complexity, it takes O(k^3) steps to terminate, where k is the number of bits, given that the number of iterations is O(k) and that divisions, etc. cost O(K^2) steps. O(k) iterations can be proved by observing that every 2 steps we at least halve the value d.

Assume that after one step this is not true, so c mod d > d/2. In the next step, we will compute d mod (c mod d) = d-(c mod d) < d/2 (i.e., d-r(c mod d), with (c mod d) > d/2 → r=1) giving the thesis.

We know that halving leads to a logarithmic complexity, i.e., linear with respect to the number k of bis.

Extended Euclidean algorithm

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 #integer division
   c-qd #this is c%d
```

Example 41. 12 mod 5 = 2, q=12/5=2, tmp=12-2*5=2.

We add two extra variables `e` and `f` to the algorithm and we save the initial value of d in `d0` 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
```

`EuclidExt(c,d)` means that we want to compute the inverse of `c` modulo `d`.

Example 42. Let's compute `EuclidExt(5,17)`:

- q=0, tmp=5, c=17, d=5, tmp=1, e=0, f=1;
- q=3, tmp=2, c=5, d=2, tmp=-3, e=1, f=-3;
- q=2, tmp=1, c=2, d=1, tmp=7, e=-3, f=7;
- q=2, tmp=0, c=1, d=0, tmp=-17 e=7, f=-17.

e%d0=7, so e=7 is the inverse, we can see that 7*5 mod 17 = 1.

Exercise 34. Try to compute `EuclidExt(14,17)`.

- q=0, tmp=14, c=17, d=14, tmp=1, e=0, f=1;
- q=1, tmp=3, c=14, d=3, tmp=-1, e=1, f=-1;
- q=4, tmp=2, c=3, d=2, tmp=5, e=-1, f=5;
- q=1, tmp=1, c=2, d=1, tmp=-6, e=5, f=-6;
- q=2, tmp=0, c=1, d=0, tmp=17, e=-6, f=17;

e%d0=11, so 11 is the inverse, 11*14 mod 17 = 1.

In practice, usually the public exponent b is fixed and takes the values of the prime number 2^{16}+1=65537. If we use at least 3072 bits, it is secure enough.

If an attacker knows ϕ(n) and n, he can crack everything by solving the system of equation:

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

Exercise 35. Decrypt the following ciphertext, which was encrypted using a Shift Cipher:

```
BEEAKFYDJXUQYHYJIQRYHTYJIQFBQDUJIIKFUHCQD
```

Using k=16

```
LOOK UP IN THE AIR IT S A BIRD IT S A PLANETS SUPERMAN
```

Exercise 36. Compute `EuclidExt(17,5)`

- q=3, tmp=2, c=5, d=2, tmp=1, e=0, f=1;
- q=2, tmp=1, c=2, d=1, tmp=-2, e=1, f=-2;
- q=2, tmp=0, c=1, d=0, tmp=5, e=-2, f=5;

e%d0=3, so 3 is the inverse, 3*17 mod 5 = 1.

Exercise 37. We intercept a 'strange' ciphertext 0x000..000115E314E61F9 (i.e., lots of 0s at the beginning) that we know it has been encrypted with RSA under a key of 2048 bits long modulus. What can we immediately conclude about the cipher public key, with very high probability, and how can we trivially compute the plaintext?

The fact that we have a lot of zeros makes us think that the original plaintext was much smaller than the modulus. In other words n is very big and x is so small that encrypting it we do not use the modulus as it is already <n. As we do not used the modulus, we can just do the b-th root of the ciphertext to obtain the original plaintext.

Lecture 17

In RSA we need to generate the two big primes **p** and q. How can we?

The easiest (but not so efficient) solution is to use what is called the "**Sieve of Eratosthenes**". This algorithm allows us to find all the prime numbers less or equal than N and works as follows: we pick all numbers from 2 to N, starting from the smallest (2, initially) that we call p, we remove from the list all of its multiples $2p$, $3p$, ... We iterate the procedure by picking the smallest p survived from the previous steps.

Example 43. Let's consider N=10, we will have the list {2,3,4,5,6,7,8,9,10}.
Starting from 2 we remove 4,6,8,10, obtaining {2,3,5,7,9}.
Consider 3 and remove 9, obtaining {2,3,5,7}.
Consider 5 and remove nothing.
Consider 7 and remove nothing.

Thus te the prime numbers remaining in the list are {2,3,5,7}.

It is not needed to check every time all the multiplies, for example, if I am considering p=3, I can skip 6 as because it is a multiple of 2, it has already been deleted when I was considering p=2. For this reason I can skip all the multiplies up to $p(p\text{-}1)$, so I can start checking from p^2. Obviously if p^2 is out of my sequence I can stop the list scanning, so the last item N of the list will be checked from $\sqrt{N}$, that means that I stop scanning after the p=$\sqrt{N}$ element of the list.

A simple python implementation is the following:

```python
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 than p
        sieve.discard(q)
  print sieve                       #sieve has only primes
```

Example 44. Sieve(15)=?
Ns=Int(Sqrt(15))=3, we take the lower part of the square root.
sieve={2,3,4,5,6,7,8,9,10,11,12,13,14,15}
p={2,3}
p=2 in sieve, q={2^2=4,..,15}

sieve={2,3,5,7,9,11,13,15}
p=3 in sieve, q={3^2=9,..,15}
Sieve(15)={2,3,5,7,11,13}

Exercise 38. Sieve(22)=?
Ns=Int(Sqrt(22))=4, we take the lower part of the square root.
sieve={2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22}
p={2,3,4} p=2 in sieve, q={2^2=4,..,22}
sieve={2,3,5,7,9,11,13,15,17,19,21} p=3 in sieve, q={3^2=9,..,22}
sieve={2,3,5,7,11,13,17,19}
p=4 not in sieve
Sieve(22)={2,3,5,7,11,13,17,19}

There is a bed news, unfortunately this algorithm takes at least N steps, meaning that it takes 2^{3072} steps for the smallest RSA modulus, that is too much. We can use another approach, we generate two random numbers and then we check if they are primes or not.

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 "**PRIMALIY is in P**" 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>3072.

Miller-Rabin test

We illustrate one of the most famous probabilistic primality test, due to Miller and Rabin. The algorithm is a **NO-biased Montecarlo**, i.e., it is always correct for the *NO* answer (meaning that the number is not prime) but can be wrong for the *YES* case (the number is prime). The probability of being wrong, however, is $\leq$ than a constant $\epsilon = \frac{1}{4}$.

We can repeat the test until the error is small enough. If we run the test r times, the probability that it says *YES* and that the number is not prime is less then $\epsilon^r = (\frac{1}{4})^r$. This means that 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 $1/4^{128}$, which is less than brute-forcing a typical symmetric key, and so small to be negligible.

The pseudocode of the algorithm is in the following chunk.

```
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
```

We can always write n-1 as 2^k*m as we will not check the primality for even numbers, but only for odd ones. So if I check the primality of an odd number *n*, I can write the even number *n-1* as 2^k*m.

Example 45. We want to compute `Prime(7)`:
n-1=6, it is even and we decompose it as 2^1*3. So we have k=1 and m=3.
We compute b as a^m mod n, we choose a=2 (1<a<n), so b=2^3 mod 7=1.
b=1 so the algorithm returns `True`.

In case we choose a=3, we have b=3^3 mod 7=6 and also in this time the algorithm returns `True`, entering the loop.

Exercise 39. Try to compute `Prime(15)`:
n-1=14, we can decompose it as 2^1*7, having k=1 and m=7.
We choose a=2 and we compute b=2^7 mod 15=8.
We enter the loop (i=0), b is not equal to n-1 so we compute the new b as b=8*8 mod 15=4.
(i=1) is already too big to stay in the loop so we exit and return `False`.

Exercise 40. Try to compute `Prime(9)`:
n-1=8, we can decompose it as 2^3*1, so k=3 and m=1.
We choose a=2 and we compute b=2^1 mod 9=2.
We enter the loop (i=0), b is not equal to n-1 so we compute the new b as b=2*2 mod 9=4.
(i=1) b is different from n-1, the new b is b=4*4 mod 9=7.
(i=2) b is different from n-1, the new b is b=7*7 mod 9=4.
(i=3) is too big to stay in the loop so we exit and return `False`.

Theorem 4. The above algorithm is always correct on False answers (NO-biased).

Proof. Let's assume, by contradiction, that the algorithm returns False, but the number n is prime. False is only returned at the last step, which means that the algorithm "survived" all the if branches. In particular we have that a^m mod n $\neq$1. Moreover a^{m2^i} mod n $\neq$n-1, for i=0,...,k-1 (for k cycles b=b*b mod n and b $\neq$n-1).

By the assumption that n is prime and by the Euler theorem (a and n are coprime since a<n and n is prime) we know that $a^{\phi(n)}$ mod n = a^{n-1} mod n = a^{m2^k} mod n=1. This is true because, as n is prime, ϕ(n)=n-1 and we know from the algorithm that we can write n-1=2^k*m.

In summary we have:

$$a^m mod\ n \neq 1, n-1$$
$$a^{2^1 m} mod\ n \neq n-1$$
$$a^{2^2 m} mod\ n \neq n-1$$
$$\ldots$$
$$a^{2^{k-1} m} mod\ n \neq n-1$$
$$a^{2^k m} mod\ n = n-1$$

The last condition is given by the Euler theorem.

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 iff r^2 mod n=1, i.e., r^2=z*n+1 for some z, meaning that r^2-1=(r+1)(r-1)=z*n which implies (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.

Notice that, if n is not prime there might be square roots of 1 different form 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 root of the preceding one. Since the last one (a^{m2^k}) is equal to 1, we have that the previous ($a^{m2^{(k-1)}}$) 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 that it is 1. We iterate backward this on all the values until we got 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 is the following:

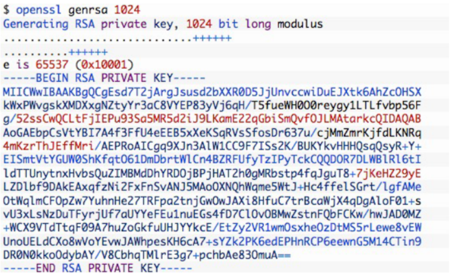
```
def Prime(n):
  m = n-1
  k = 0
  while (m&1 == 0):      #while m is even
    m = m>>1
    k += 1
  a = random.randrange(2,n)
  b = squareAndMultiply(a.m,n)
  if b == 1:
    return True
  for i in range (k):
    if b == n-1:
      return True
    b = b*b % n
  return False
```

So, how can we 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. For example, if we want 512 bits of size (for a modulus 1024), we have that ln(n) = $\log_2$(n)*ln(2) ≈ 512*0.7 ≈ 358. Thus, on average, after 358(716) attempts we should find a prime number of size 512 (1024) bits.

Notice that, the Miller-Rabin algorithm should be iterated to lower as needed the probability of error in case of successful answer. In the original paper, the probability is bounded by 1/4, in a subsequent paper it has been shown that the algorithm is much more precise than that. In practise the suggested number of iterations from NIST are 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.

Case study: OpenSSL

OpenSSL is a software library for applications that need to secure communications over the internet widely used against eavesdropping and 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. It supports a lot of different cryptographic algorithms: *ciphers* (such as AES, Blowfish, DES, IDEA, ..), *cryptographic has functions* (MD5, SHA-1, ...) and *public-key cryptography* (RSA, DSA, ...).



```
$ openssl genrsa 1024
Generating RSA private key, 1024 bit long modulus
..............................++++++
..........++++++
e is 65537 (0x10001)
-----BEGIN RSA PRIVATE KEY-----
MIICWwIBAAKBgQCgEsd7T2jArgJsusd2bXXR0D5JjUnvccwiDuEJXtk6AhZcOHSX
kWxPWvgskXMDXxgNZtyYr3aC8VYEP83yVj6qH/T5fueWH0O0reygy1LTLfvbp56F
g/52ssCwQCLtFjIEPu93Sa5MR5d2iJ9LKamE22qGbiSmQvfOJLMAtarkcQIDAQAB
AoGAEbpCsVtYBI7A4f3FfU4eEEB5xXeKSqRVsSfosDr637u/cjMmZmrKjfdLKNRq
4mKzrThJEffMri/AEPRoAICgq9XJn3AlW1CC9F7ISs2K/BUKYkvHHHQsqQsyR+Y+
EISmtVtYGUW0ShKfqtO61DmDbrtWlCn4BZRFUfyTzIPyTckCQQDOR7DLWBlRl6tI
ldTTUnytnxHvbsQuZIMBMdDhYRDOjBPjHAT2h0gMRbstp4fqJguT8+7jKeHZ29yE
LZDlbf9DAkEAxqfzNi2FxFnSvANJ5MAoOXNQhWqme5WtJ+Hc4ffelSGrt/lgfAMe
OtWqlmCFOpZw7YuhnHe27TRFpa2tnjGwOwJAXi8HfuC7trBcaWjX4qDgAloF01+s
vU3xLsNzDuTFyrjUf7aUYYeFEu1nuEGs4fD7ClOvOBMwZstnFQbFCKw/hwJAD0MZ
+WCX9VTdTtqF09A7huZoGkfuUHJYYkcE/EtZy2VR1wmOsxheOzDtMS5rLewe8vEW
UnoUELdCXo8wVoYEvwJAWhpesKH6cA7+sYZk2PK6edEPHnRCP6eewnG5M14CTin9
DR0N0kkoOdybAY/V8CbhqTMlrE3g7+pchbAe83OmuA==
-----END RSA PRIVATE KEY-----
```

Figure 36: Generation of a Private Key using OpenSSL.

In Figure 36 we can see the generation of a Private Key using OpenSSL. It uses a number of iterations slightly bigger than the one suggested above. For a 1024 key as in figure we have 6 occurrences of '+', which represent 6 iterations of the Miller-Rabin test.

Lecture 18

As already said, security of RSA is based on the secrecy of ϕ(n)=(p-1)(q-1). We know that if n is big enough, its factorization is a hard problem so that the attacker cannot go back to p and q. It is however trivial to see that computing ϕ(n) is at least as difficult as factoring n. Given an algorithm that computes ϕ(n), to compute p and q it is enough to solve the system of equations:

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

We can make another observation: it could be possible, in principle, to compute the private exponent without necessarily computing ϕ(n). It can be proved that this would allow to factorize n.

Theorem 5. Given an algorithm that computes the exponent a, we can write a probabilistic "Las Vegas" algorithm that factorizes n with probability at least 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 factorizing. 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.

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 will now discuss some attacks that are possible if we choose an excessively small exponent such as 3.

Attack 1: same message encrypted under different modules

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. Notice that, modules n_1, n_2, n_3 are very likely to be coprime as they will not share their prime factors.

Theorem 6. Chinese Remainder Theorem: Let n_1, ..., n_k be integers >1, and N=n_1*n_2*...*n_k. If the n_i are pairwise coprime, and if a_1, ..., a_k are integers such that 0≤a_i ≤n_i for every i, then there is one and only one integer x such that 0≤x≤N and x mod n_i=a_i for every i.

The Chinese Remainder 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. In our case x will be m^3.

Notice that m<n_i, which implies m^3<n_1*n_2*n_3. By definition of c_i we also have c_i=m^3 mod n_i. Thus, the unique x given by the Chinese Reminder Theorem must be equal to m^3. It is now enough to compute $(x)^{1/3}$ to get m.

Example 46. Let's consider m=2, n_1=3, n_2=5, n_3=7.
$c_1 = m^3 \bmod n_1 = 8 \bmod 3 = 2$
$c_2 = m^3 \bmod n_2 = 8 \bmod 5 = 3$
$c_3 = m^3 \bmod n_3 = 8 \bmod 7 = 1$

The attacker sees the three ciphertexts 2,3,1.

The Chinese Reminder Theorem tells us that ∃x: x<3*5*7
Moreover it is true that $0 \leq c_i < n_i$ for all i. Thus it is true that there exists a unique x such that x mod n_i=c_i.

We will get that x=8, thus x=m^3=8, so m=$\sqrt[3]{8}$=2 that is our message.

We can say that, 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, in fact we 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 (1 message here is enough). Consider a small message m encrypted with exponent 3. we have then y=m^3 mod n and it might be the case that m^3<n meaning that y=m^3 mod n=m^3. To decrypt it is enough to compute $y^{1/3}$. Padding prevents this attack by making the size of m^3 close to the size of the modulus n.

Partial information

It could be possible that an attacker is able to partially recover the plaintext. This attack has stronger assumptions but so it is a bit harder to run. Let y=E_{PK}(x).

Examples of partial information are:

- parity(y): returns the parity of the plaintext;
- half(y): tells if the plaintext is less than half of the modulus n.

We can define them as:

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

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

We claim that if an attacker knows these information, he is able to decrypt the ciphertext. First, we show that the latter can be defined based on the former. This means that an attacker needs only one of the two information to decrypt the message.

The result exploits the following fundamental property of RSA.

Theorem 7. RSA multiplicative property: RSA cipher is such that $\mathrm{E}_{PK}(\mathrm{x}_1)\mathrm{E}_{PK}(\mathrm{x}_2)$ mod n=$\mathrm{E}_{PK}(\mathrm{x}_1\mathrm{x}_2$ mod n).

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

$$\begin{aligned} E_{PK}(x_1)E_{PK}(x_2) \text{ mod n} &= x_1^b x_2^b \text{ mod n} \\ &= (x_1x_2)^b \text{ mod n} \\ &= (x_1x_2 \text{ mod n})^b \text{ mod n} \\ &= E_PK(x_1x_2 \text{ mod n}) \end{aligned}$$

■

We can now prove that

half(y)=parity(E_{PK}(2)y mod n).

Proof. Notice that, parity(E_{PK}(2)y mod n)=parity(E_{PK}(2)E_{PK}(x) mod n)=parity(E_{PK}(2x mod n)) for the previous property.

If half(y)=0, by definition we have 0≤x<n/2 which implies 0≤2x<n, thus 2x mod n=2x which is certainly even, i.e., parity(E_{PK}(2)y mod n)=parity(E_{PK}(2x mod n))=0 (by definition of parity).

If 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 parity(E_{PK}(2)t mod n)=parity(E_{PK}(2x mod n))=1.

■

At this point, the following holds:

Theorem 8. Given al 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), 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(yE_{PK}(x)). It is 0 when 2x mod n<n/2 which happens when 0≤x<n/4 or n/2≤x<3n/4, since we respectively obtain 0≤2x<n/2 and n≤2x<3n/2, the latter implying n mod n≤2x mod n<3n/2 mod n, i.e., 0≤2x mod n<n/2.

This reasoning can be applied, similarly, to next steps until a single solution survives. Since each time the interval is split in two, the number of steps is $\lceil \log_2(n) \rceil$. ■