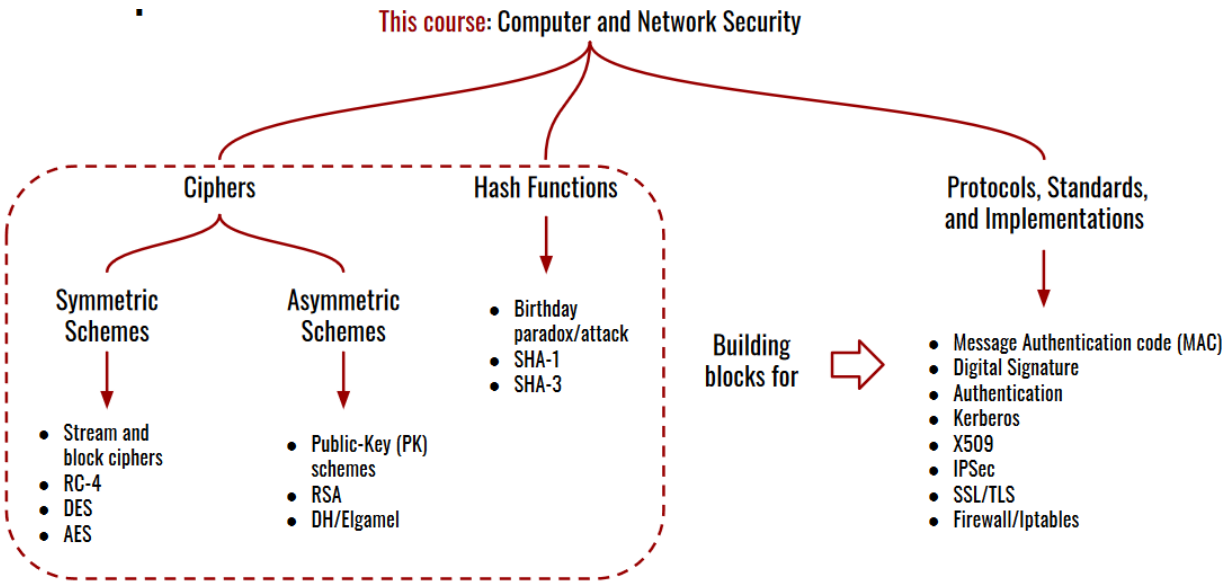
COMPUTER AND NETWORK SECURITY

Exam Notes



Index

Basics

Security: freedom from, or resilience against, potential harm (or other unwanted coercive change) caused by others.

Safety: prevents unintentional accidents.

Security Blocks

Main building blocks to offer Security Services, defined by several standards, e.g. ISO/IEC 27000:2018, CNSS glossary, ISACA.

- **Confidentiality:** information is kept secret from all except authorized parties.
- **(Data) Integrity:** information is not tampered while in transit. Crucial for ensuring that data has not been altered by a third party.
- **Availability:** the system or the information is reliably available to the end users

These three properties above are called CIA.

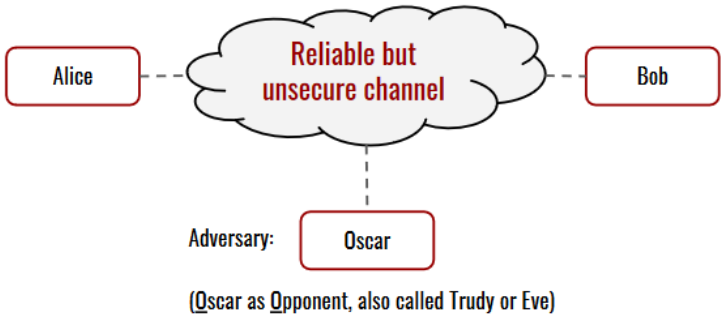
- **Message Authentication**: the sender/creator of the message is authentic. Often includes integrity of the message. It is offered with Message Authentication Code, Digital Signature, Authenticated Encryption (AE). It is the assurance that a given entity was the original source of the received data.
- **Entity Authentication**: establish and verify the identity of an entity. Passwords are a good example, also single and multi-factor authentication are. It is the assurance that a given entity is involved and currently active in a communication session.
- **Non Repudiation**: the sender of a message cannot deny the creation of the message.

Terminology*

- encryption function (and algorithm): E
- decryption function (and algorithm): Dor E^{-1}
- encryption key: k_1
- decryption key: k_2
- key space: number of bits used for keys
- message space: number of bits used for message
- $m = D_{k_2}(E_{k_1}(m))$
- when $k_1 = k_2$then we have a **symmetric** scheme
- when $k_1 \neq k_2$ then we have an **asymmetric** scheme
- Message m is also called plaintext, encrypted m is also called ciphertext

* Sometimes terminology may change for practical reasons. It will be stated first.

Communication and Adversary Models



We assume that the adversary knows algorithms and protocols (no security through obfuscation) and also message and key space.

Kerckhoffs' Principle. A cryptosystem should be secure even if the attacker knows all details about the system, with the exception of the secret key. In particular, the system should be secure when the attacker knows the encryption and decryption algorithms.

- **Eavesdropping**: secretly listening to private conversation of others without their consent
- **Known Plaintext**: attacker has samples of both plaintext and its encrypted version (ciphertext) and is at liberty to make use of them to reveal further secret information such as secret keys.
- **Chosen Plaintext**: attacker has the capability to choose arbitrary plaintexts to be encrypted and obtain the corresponding ciphertexts. The goal of the attack is to gain some further information which reduces the security of the encryption scheme. In the worst case, a chosen-plaintext attack could reveal the scheme's secret key.
- **Adaptive Chosen Plaintext**: the cryptanalyst makes a series of interactive queries, choosing subsequent plaintexts based on the information from the previous encryptions.
- **Chosen Ciphertext**: the cryptanalyst gathers information, at least in part, by choosing a ciphertext and obtaining its decryption under an unknown key.
- **Physical Access**

Common attacks are:

- **Replay:** a valid data transmission is maliciously or fraudulently repeated or delayed.
- **Reflection:** some protocols are based on 'challenge-response' authentication and this attack attempts to trick an entity into proving the right answer to its own challenge.
- **Man-in-the-Middle (MITM):** attacker secretly relays and possibly alters the communications between two parties who believe that they are directly communicating with each other.

EXAMPLE Exam 2015 Feb 10th

Q2.1 : describe the attack Man-In-The-Middle, as well as a possible scenario where it could be run.

A MITM attack is when an attacker secretly relays and possibly alters the communications between two parties who believe that they are directly communicating with each other.
An example is **active eavesdropping**, where the attacker Trudy makes independent communications with the victims Alice and Bob and relays messages between them to make them believe they are communicating directly with each other over a private connection while all the conversation is actually **controlled by the attacker**.
A typical MITM scenario is when the attack is performed over a standard Diffie Hellman key exchange, where Trudy can obtain the secret key of both Alice and Bob, who will think to speak to each other.

Q2.2 : Alice suspects she is currently being the target of a Man-In-The-Middle attack, and she decides to hire you as a personal adviser.Can she still carry out safe actions on the Internet? Discuss.

To avoid being victim of a MITM attack Alice should communicate using security precautions to ensure **authentication**. This kind of property is offered by MACs or Digital Signature schemes.
In the case of use of **MACs**, she should first find a **secure way to exchange the key** with whom she wants to communicate with (Bob). Since Diffie Hellman key exchange can be compromised, a good method could be to encrypt a message containing the MAC key with Bob's public key (maybe using RSA) and then send the message to Bob. Doing so, the attacker will not be able to read the message and so she can start a more secure communication with Bob using MACs.
If Alice wants to avoid for any reason the step of sending the key to Bob encrypted with RSA, another path to achieve authentication is with **Digital Signatures**. By only communicating with signed messages, Alice can be sure about the origin of incoming messages, so that she can verify whether the sender is really Bob by verifying his sign with his public key.

Mathematical Background I

Basilar mathematical concepts behind symmetric cryptography and AES.

Modular Arithmetic

Let $a, m, r \in Z$ with $m > 0 : a = r \, mod \, m$
m divides $(a - r)$, m is called **modulus**, r is called **remainder**.
Additionally we can write $a = q * m + r$, where q is called **quotient**.
The remainder is **not unique**:

- $12 = 3 \, mod \, 9 \, , q = 1$
- $12 = 21 \, mod \, 9, q = -1$
- $12 = -6 \, mod \, 9, 1 = 2$

By convention, we always choose the smallest non-negative remainder.
The set of all remainders form an **equivalence class** modulo m, meaning that they behave equivalent modulo m.

Congruence - Two natural numbers a, b are said to be congruent modulo n if $a = b \, mod \, n$.
The integer division of (a and n) and (b and n) yield the same remainder. The congruence relation is **reflexive**, **symmetric** and **transitive**, hence it is an equivalence relation.

Properties:

- invariance over addition: $a \equiv b \bmod n \Leftrightarrow (a + c) \equiv (b + c) \bmod n$
- invariance over multiplication: $a \equiv b \bmod n \Leftrightarrow (a \cdot c) \equiv (b \cdot c) \bmod n$
- invariance over exponentiation: $a \equiv b \bmod n \Leftrightarrow a^k \equiv b^k \bmod n$

Modular Division

$$b/a \equiv b \cdot a^{-1} \bmod m$$

The inverse of a is defined as:

$$a \cdot a^{-1} \equiv 1 \bmod m$$

E.g., $5/7 \equiv 5 \cdot 4 = 20 \equiv 2 \bmod 9$ since $7 \cdot 4 = 28 \equiv 1 \bmod 9$

The inverse of a number a mod m exists only if:

$$gcd(a, m) = 1$$

To compute the greatest common divisor (gcd), we can use the Euclidean algorithm.

Algebraic Structures

Given a set:

- **Group:** addition and its inverse
- **Ring:** addition, additive inverse, multiplication
- **Field:** addition, additive inverse, multiplication, multiplicative inverse.

A field is basically a structure in which we can always add, subtract, multiply and divide.

Group

A group G is a set of elements and one group operator "$\circ$" such that:

1. closure: $\forall a, b \in \mathrm{G} : a \circ b \in \mathrm{G}$
2. associativity $\forall a, b, c \in \mathrm{G} : (a \circ b) \circ c = a \circ (b \circ c)$
3. neutral element $\exists\, 1 \in \mathrm{G}$ such that $\forall a \in \mathrm{G} : a \circ 1 = a$
4. inverse element $\forall a \in \mathrm{G}\,, \exists\, a^{-1} \in \mathrm{G} : a \circ a^{-1} = 1$

If abelian group then also:

5. commutative $\forall a, b \in \mathrm{G} : a \circ b = b \circ a$

The group operator is often referred, by convention, with the operator "+".
If we add another group operator, by convention, we refer to it with the operator "*".

An **abelian group**, also called a **commutative group**, is a group in which the result of applying the group operation to two group elements does not depend on the order in which they are written. In other words, the group operation is **commutative**.

SUBGROUP - If G is a group, you can obtain what is called a subgroup by making some subset of G, named H, by taking the same operations. Generally speaking, it's not true that taking some elements of a group and the same operations, what you get is still a group. If this happens, you get a subgroup. For example, the subset could not be closed under an operation.

Ring

A ring R is a set with two operations "addition" and "multiplication" such that:

$$\forall a, b, c, d \in \mathrm{R} : \begin{array}{l} a + b = c \in \mathrm{R} \\ a \cdot b = d \in \mathrm{R} \end{array}$$

Hence, there is closure (the result is in the ring). Additionally:

- addition and multiplication are associative
- addition is commutative
- distributive law holds
- there exists a neutral element for addition
- there exists always an additive inverse for addition given an element in the ring
- there exists a neutral element for multipl.
- the multiplicative inverse exists only for some elements but not for all.

- abelian group under addition:
  - closure: $\forall a, b \in \mathrm{R} : a + b \in \mathrm{R}$
  - addition is associative $\forall a, b, c \in \mathrm{R} : (a + b) + c = a + (b + c)$
  - addition is commutative $\forall a, b \in \mathrm{R} : a + b = b + a$
  - neutral element for addition $\exists\, 0 \in \mathrm{R}$ such that $\forall a \in \mathrm{R} : a + 0 = a$
  - inverse for addition $\forall a \in \mathrm{R}\,, \exists\, -a \in \mathrm{R} : a + (-a) = 0$
- monoid group under multiplication:
  - closure $\forall a, b \in \mathrm{R} : a \cdot b \in \mathrm{R}$
  - multiplication is associative $\forall a, b, c \in \mathrm{R} : (a \cdot b) \cdot c = a \cdot (b \cdot c)$
  - multiplication is commutative $\forall a, b \in \mathrm{R} : a \cdot b = b \cdot a$
  - distribution law holds $\forall a, b, c \in \mathrm{R} : a \cdot (b + c) = a \cdot b + a \cdot c$
  - neutral element for multiplication $\exists\, 1 \in \mathrm{R}$ such that $\forall a \in \mathrm{R} : a \cdot 1 = a$

Informally, a ring is a structure in which we can always add, subtract and multiply, but we can only divide by certain elements (namely by those for which a multiplicative inverse exists).

We define the "integer ring" $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$ where we perform modular arithmetic by m:

E.g., $\mathbb{Z}_9 = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$

Coprime or relatively prime

An element has a **multiplicative inverse** if and only if: $gcd(a, m) = 1$.
We say that a is **coprime** or **relatively prime** to m.
The inverse can be found with the **Extended Euclidean Algorithm**.

Field

A field F is a ring where every non negative element has a multiplicative inverse in the field:

$$\forall a \in \mathrm{F}, \exists\, a^{-1} \in \mathrm{F} : a \cdot a^{-1} = 1$$

Element 0 does not need to have the inverse.
A finite field (also called Galois Field) is a field where the set F is finite.

E.g., $\mathbb{Z}_9 = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ is not a field.

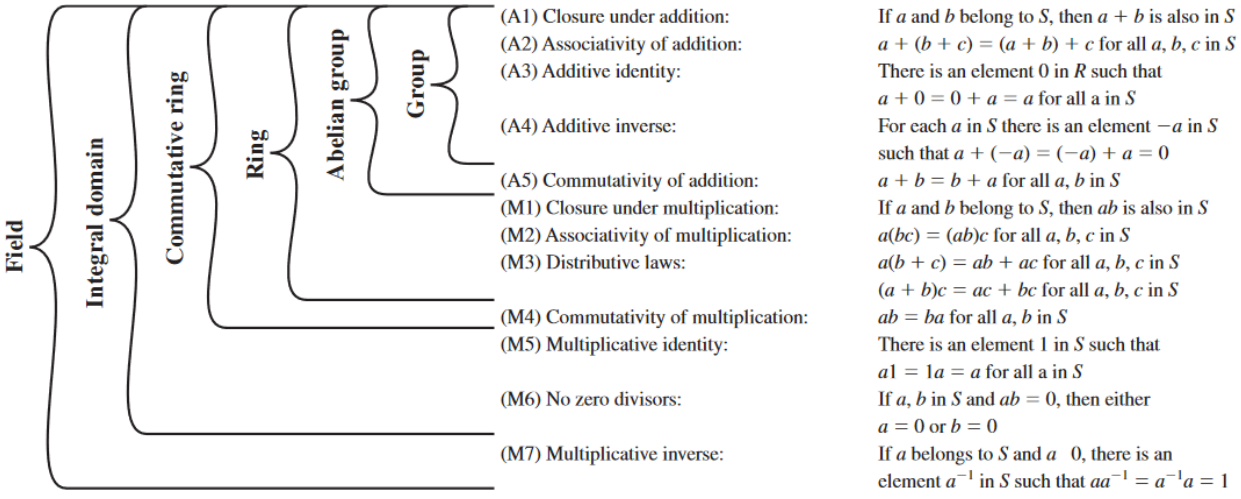
$\mathbb{Z}_5 = \{0, 1, 2, 3, 4\}$ is a field.

Theorem. Let F be a field, then for any non negative element a the inverse a^{-1} is unique.

Proof by contradiction. Suppose a^{-1} and b are two inverses of a then:

$ab = 1$
$a^{-1} ab = 1\, a^{-1}$
$(a^{-1} a)b = 1\, a^{-1}$
$1\, b = 1\, a^{-1}$
$b = a^{-1}$ (hence the two inverses must be equal and cannot be different)

This property can be exploited in cryptography to implement a scheme based on the relation between an element and its inverse.



Finite Fields (or Galois Fields)

Theorem. It can be proved that $\mathbb{Z}_p$ is a field if p is prime.
Theorem. Finite fields exist only if they have p^m elements, where p is a prime and m an integer.

E..g.,

There exists a finite field with 2 elements (called GF(2)): p = 2, m = 1
There exists a finite field with 11 elements (called GF(11)): p = 11, m = 1
There is NOT a finite field with 12 elements: $12 = 3 * 4 = 3 * 2^2$
There exists a finite field with 81 elements (called $GF(81) = GF(3^4)$): p = 3, m = 4
There exists a finite field with 256 elements (called $GF(256) = GF(2^8)$): p = 2, m = 8

$GF(2^8)$ is used by AES.

Two types of $GF(p^m)$:

- m = 1 : prime fields, written as GF(p)
- m > 1: extension fields

They are both used extensively in cryptography. In particular, Galois Fields with p=2, i.e., $GF(2^m)$, are very common and well studied.
The elements of a prime field GF(p) are the integers {0, 1, 2, ..., p-1}

- Addition: $a + b \equiv c \bmod p$
- Subtraction: $a - b \equiv d \bmod p$
- Multiplication: $a \cdot b \equiv e \bmod p$

Hence, for {addition, subtraction, multiplication} is just the operation modulo p. Properties of the ring are satisfied when performing arithmetic in this way.

Euclidean Algorithm

Useful to find the **greatest common divisor** for two numbers a, b : **gcd(a, b)**.
Usage of a table to calculate gcd(a, b), where **q = quotient = a//b** and **r = remainder = a - (b*q)**

Example: find gcd(36, 10)

a	**b**	**q**	**r**
36	10	3	6
10	6	1	4
6	4	1	2
4	2	2	0

The algorithm ends when r=0.
Once wrote down a row, the values of a and be for the next row are:
a = b of the previous row
b = r of the previous row
The answer of the calculation is the value of **b** in the last row, so in this case is **gcd(36, 10) = 2**.

Mathematical Background II

Basilar mathematical concepts behind asymmetric cryptography and RSA.

Euler's Phi Function (or Totient Function)

Given $Z_m = \{0, 1, ..., m-1\}$, then the function $\Phi(m)$ is the number of how many elements in Z_m are coprime to m, i.e. how many numbers a exist in the given set such that $gcd(a, m) = 1$.
For a prime number p, we can say that $\Phi(p) = (p-1)$.

Euler's Theorem

Let a and m be integers with $gcd(a, m) = 1$. Then:

$$a^{\Phi(m)} = 1 \bmod m \ .$$

Example

Compute:

$$5^{200} \bmod 23$$

Since 23 is prime then:

$$\Phi(23) = 22$$
$$200 = 9 \cdot 22 + 2$$
$$5^{200} = (5^{22})^9 \cdot 5^2$$
$$5^{\Phi(23)} = 5^{22} \equiv 1 \bmod 23$$
$$5^{200} \equiv 1^9 \cdot 5^2 \bmod 23 \equiv 25 \bmod 23 \equiv 2 \bmod 23$$

Example

Compute:

$$2^{280} \bmod 251$$

Since 251 is prime then:

$$\Phi(251) = 250$$
$$280 = 1 \cdot 250 + 30$$
$$2^{280} = 2^{250} \cdot 2^{30}$$
$$2^{\Phi(251)} = 2^{250} \equiv 1 \bmod 251$$
$$2^{280} \bmod 251 = 1 \cdot 2^{30} \bmod 251$$
$$2^{30} = 2^{10} \cdot 2^{10} \cdot 2^{10} = 1024 \cdot 1024 \cdot 1024 \equiv 20^3 \bmod 251 = 219 \bmod 251$$

Fermat's Little Theorem - Multiplicative Inverse

If p is prime and a is a positive integer not divisible by p, then:

$$a^p = a \bmod p \Rightarrow a^{p-1} = 1 \bmod p \Rightarrow a^{-1} = a^{p-2} \bmod p$$

ant it is useful in some cases to finde the multiplicative inverse instead of using the Extended Euclidean Algorithm (EEA).

We can use this theorem in some cases to compute the multiplicative inverse, instead of using EEA.

E.g.,

$$a^{p-2} = 2^5 = 32 \equiv 4 \bmod 7$$

4 is indeed the multiplicative inverse of 2 modulo 7

Example: What is the Multiplicative Inverse of 3 in Z_{10}? It ia 7, because 3 * 7 = 21, and 21 is 1 in Z_{10}.

Example: What is the Multiplicative Inverse of 8 in Z_{10}? It **does not exist**, since an integer x has a Multiplicative Inverse iff gcd(x, n) = 1, and so 8 has no multiplicative inverse in Z_{10}.

In modular arithmetic mod n, the Multiplicative Inverse of x is the integer y such that $(x \times y)\, mod(n) = 1\, mod(n)$.

Chinese Remainder Theorem (CRT)

In essence, the CRT says it is possible to reconstruct integers in a certain range from their residues modulo a set of pairwise relatively prime moduli.[2] Given p and q coprime, if:

$$x = a \bmod p \quad \text{and} \quad x = a \bmod q$$

Then :

$$x = a \bmod pq$$

More about Groups

Order of a Group

Given a finite group G (i.e. a group with a finite number of elements), the number of elements in the group (**cardinality**) is called "order of the group".

$$| G | = \text{"order of the group"}$$

Multiplicative Group

$Z*_m$ is a Multiplicative Group if it is the set of positive numbers smaller than m that are **relatively coprime** to m. The cardinality it thus equal to $\Phi(m)$.

Remember: An element has a **multiplicative inverse** if and only if: $gcd(a, m) = 1$.
We say that a is **coprime** or **relatively prime** to m.
The inverse can be found with the **Extended Euclidean Algorithm**.

Order of an element and Generators

The order of an element $a \in G$ is the smaller positive integer k such that $a^k = 1$.
Elements with maximum order $ord(a) = \mid G \mid$ are called **primitive elements** or **generators**.
A group G which contains an element with maximum order is said to be **cyclic**.
For every prime p, it can be proved that the Abelian group $(z*_p, \cdot)$ is a cyclic group, i.e. a multiplicative group of every prime field is cyclic.
A generalization of Fermat's Little Theorem tells that $a^{|G|}$ is a finite cyclic group.

Discrete Logarithm Problem (DLP)

Given the finite cyclic group $Z*_p$ of order $p - 1$ and two primitive elements a, b for the group, the DLP is the problem of determining the integer $1 \leq x \leq p - 1$ such that $a^x = b \bmod p$.

Bézout's Identity

Let a and b be integers with greatest common divisor d s.t. $gcd(a, b) = d$.
Then there exist integers x and y such that $ax + by = d$.
More generally, the integers of the form $ax + by$ are exactly the multiples of d.

Extended Euclidean Algorithm (EEA)

Computes, in extension to the Euclidean Algorithm, also the coefficients for Bézout's identity.
Assume a and b are known: we want to compute the values of s and t such that $as + tb = gcd(a, b)$.
EEA also computes the multiplicative inverse when $gcd(a, b) = as + tb = 1$.
$0 \times a + t \times b = 1 \, mod \, a$, and t is the inverse we want to find.
Here is used a different terminology where $a = r_0, b = r_1$ s.t. we want to find $gcd(r_0, r_1) = s \times r_0 + t \times r_1$.

Three main steps:

1. Compute $gcd(r_0, r_1)$ using EA
2. Compute $r_0 = q_1 \cdot r_1 + r_2$
3. Compute $r_2 = s_2 \cdot r_0 + t_2 \cdot r_1$

Repeat this process recursively.

$$
\begin{array}{lll}
gcd(r_0, r_1) & r_0 = q_1 \cdot r_1 + r_2 & r_2 = s_2 \cdot r_0 + t_2 \cdot r_1 \\
gcd(r_1, r_2) & r_1 = q_2 \cdot r_2 + r_3 & r_3 = s_3 \cdot r_0 + t_3 \cdot r_1 \\
\vdots & \vdots & \vdots \\
gcd(r_{l-2}, r_{l-1}) & r_{l-2} = q_{l-1} \cdot r_{l-1} + r_l & r_l = s_l \cdot r_0 + t_l \cdot r_1
\end{array}
$$

r_l is the $gcd(r_0, r_1)$

$t_l = t$ is the multiplicative inverse iff r_l is 1

Chapter 6 of Understanding Cryptography by Christof Paar and Jan Pelzl

EXAMPLE

E.g., $gcd(973, 301) = s \cdot 973 + t \cdot 301$

$$
\begin{array}{lll}
 & \overset{r_0}{973} = 3 \cdot \overset{r_1}{301} + \overset{r_2}{70} & r_2 = 70 = 1 \cdot 973 + (-3) \cdot 301 \\
i = 2 & & \\
 & 301 = 4 \cdot 70 + \overset{r_3}{21} & r_3 = 21 = 1 \cdot 301 + (-4) \cdot 70 \\
i = 3 & &
\end{array}
$$

This is not in the form: $s \cdot 973 + t \cdot 301$

but we can replace the value of 70 with what obtained in the previous line:

$$r_3 = 1 \cdot 301 + (-4) \cdot (1 \cdot 973 - 3 \cdot 301)$$

This is what we need: $r_3 = -4 \cdot 973 + 13 \cdot 301$

36

$$i = 4 \qquad 70 = 3 \cdot 21 + \overset{r_4}{7} \qquad r_4 = 7 = 1 \cdot 70 + (-3) \cdot 21$$

This is not in the form: $s \cdot 973 + t \cdot 301$

but we can replace the values of 70 and 21 with the what obtained in the previous lines:

$$r_4 = 1 \cdot (973 - 3 \cdot 301) + (-3) \cdot (-4 \cdot 973 + 13 \cdot 301)$$
$$r_4 = 13 \cdot 973 + (-42) \cdot 301$$

Since r_5 will be zero, we have done: $t_4 = t = -42$

Remark. Since gcd(973, 301) != 1 then t is not the multiplicative inverse of 973 modulo 301.

If we generalize this process...

Chapter 6 of Understanding Cryptography by Christof Paar and Jan Pelzl

GENERALIZATION

Input: positive integers r_0 and r_1 with $r_0 > r_1$
Output: $\gcd(r_0, r_1)$, as well as s and t such that $\gcd(r_0, r_1) = s \cdot r_0 + t \cdot r_1$.

Initialization:

$s_0 = 1 \quad t_0 = 0$
$s_1 = 0 \quad t_1 = 1$
$i = 1$

Algorithm:

1 DO
1.1 $i = i + 1$
1.2 $r_i = r_{i-2} \bmod r_{i-1}$
1.3 $q_{i-1} = (r_{i-2} - r_i)/r_{i-1}$
1.4 $s_i = s_{i-2} - q_{i-1} \cdot s_{i-1}$
1.5 $t_i = t_{i-2} - q_{i-1} \cdot t_{i-1}$
WHILE $r_i \neq 0$
2 RETURN
$\gcd(r_0, r_1) = r_{i-1}$
$s = s_{i-1}$
$t = t_{i-1}$

Chapter 6 of Understanding Cryptography by Christof Paar and Jan Pelzl

Simple Examples and Exercises

PIAZZA EXERCISES (From Understanding Cryptography)

E1.5 - 1 $15 \times 29 \; mod \; 13$
$15 \; mod \; 13 = 2 \; mod \; 13$, $29 \; mod \; 13 = 3 \; mod \; 13$.
So $15 \times 29 \; mod \; 13 = 2 \times 3 \; mod \; 13 = 6 \; mod \; 13$.
When performing modulo operations imagine the numbers not as a straight line but like a clock with m numbers, where m is the value of mod.
So in the case of $29 \; mod \; 13$we see that $29 \; / \; 13 = 2$ with remainder 3, s.t. $2 \times 13 + 3 = 29$.
So it is possible to say that $29 \; mod \; 13 = 3 \; mod \; 13$ since in our imaginary clock these two numbers are in the same position.

E1.5 -2 $2 \times 29 \; mod \; 13$
$2 \; mod \; 13$ stays the same, and $29 \; mod \; 13 = 3 \; mod \; 13$.
So $2 \times 29 \; mod \; 13 = 2 \times 3 \; mod \; 13 = 6 \; mod \; 13$.

E1.5 - 4 $-11 \times 3 \; mod \; 13$
In the clock imagine to start from 0 and go backwards for 11 numbers out of 13.
Notice that $-11 \; mod \; 13 = 2 \; mod \; 13$.
So $2 \; mod \; 13 \times 3 \; mod \; 13 = 2 \times 3 \; mod \; 13 = 6 \; mod \; 13$.

E1.8 - 1 Multiplicative inverse of 5 in Z_{11}.
It is equal to ask to find the multiplicative inverse of $5 \; mod \; 11$.
First of all, an inverse exists iff $gcd(m, a) = 1$. In this case $gcd(6, 5) = 1$, so the inverse exists.
The inverse is that number y s.t. $(a \times y) \; mod \; m = 1 \; mod \; n$.
Dealing with small numbers it is easy to notice that $y = 9$, since $(5 \times 9) \; mod \; 11 = 45 \; mod \; 11 = 44 \; mod \; 11 + 1 \; mod \; 11 = 1 \; mod \; 11$. So the result is 9.

E1.8 - 2 Multiplicative inverse of 5 in Z_{12}.
It is equal to ask to find the multiplicative inverse of $5 \; mod \; 12$.
Since $gcd(12, 5) = 1$, an inverse exists.
It is easy to see that the **inverse is** $y = 5$, since $(5 \times 5) \; mod \; 12 = 25 \; mod \; 12 = 1 \; mod \; 12$.
What if we want to use the Extended Euclidean Algorithm? (*see next exercises for step by step)*

$r_0 = 12, r_1 = 5$

1. $12 = 2 \times 5 + 2$ $\quad r_0 = q_1 \times r_1 + r_2$ with $r_2 = s_2 \times r_0 + t_2 \times r_1$
2. $5 = 2 \times 2 + 1$ $\quad r_1 = q_2 \times r_2 + r_3$ with $r_3 = s_3 \times r_0 + t_3 \times r_1$

Since we have that $r_3 = 1$, **stop**.
Now from line 2 we see that $r_3 = r_1 - q_2 \times r_2$

3. $1 = 5 - 2 \times 2$.

We want to **shape** our result such that $1 = r_3 = s \times r_0 + t \times r_1$ and then take t as mult inv.

4. $1 = 5 - 2 \times 2 = 5 - 2 \times [12 - 2 \times 5] = 5 - (2 \times 12 - 4 \times 5) = 5 \times 5 - 2 \times 12$

That is in the shape we want. So pick the value that multiplies $r_1 = 5$ that is $t = 5$.

E1.8 - 3 Multiplicative inverse of 5 in Z_{13}.
It is equal to find the multiplicative inverse of $5\ mod\ 13$. An inverse exists since $gcd(13,5) = 1$.
It is easy to see that the **inverse** is $y = 8$, since $(5 \times 8)\ mod\ 13 = 40\ mod\ 13 = 1\ mod\ 13$.
If we want to apply the Extended Euclidean Algorithm:

$r_0 = 13, r_1 = 5$

1. $13 = 2 \times 5 + 3$ $\quad r_0 = q_1 \times r_1 + r_2$ with $r_2 = s_2 \times r_0 + t_2 \times r_1$
2. $5 = 1 \times 3 + 2$ $\quad r_1 = q_2 \times r_2 + r_3$ with $r_3 = s_3 \times r_0 + t_3 \times r_1$
3. $3 = 1 \times 2 + 1$ $\quad r_2 = q_3 \times r_3 + r_4$ with $r_4 = s_4 \times r_0 + t_4 \times r_1$

Since we have that $r_4 = 1$, **stop.**
Form line 2 we see that $r_4 = r_2 - q_3 \times r_3$

4. $1 = 3 - 1 \times 2$.

We want to **shape** our result such that $1 = r_4 = s \times r_0 + t \times r_1$ and take t as mult inv.

5. $1 = 3 - 1 \times 2 = [13 - 2 \times 5] - 1 \times (5 - 1 \times 3) = 13 - 2 \times 5 - 5 + (1 \times 3) =$
   $13 - 3 \times 5 + (3) =$
   $= 13 - 3 \times 5 + (13 - 2 \times 5) = 2 \times 13 - 5 \times 5$.

We are in the desired form, where $t = -5$. Now we can see that $-5\ mod\ 13 = 8\ mod\ 13$, and so we proved that the multiplicative inverse of $5\ mod\ 13$ is 8.

E1.9 - 1 Find $x = 3^2 mod\ 13$
Here it is simple, since $x = 9\ mod\ 13$

E1.9 - 2 Find $x = 7^2\ mod\ 13$
We have that $7^2 = 49$, and $49\ mod\ 13 = 10\ mod\ 13$

E1.9 - 3 Find $x = 3^{10} mod\ 13$
Now things are getting more complicated.
Since $3^3 = 27$ and $27\ mod\ 13 = 1\ mod\ 13$, we can write $3^{10} = 3^3 \times 3^3 \times 3^3 \times 3 =$
$3 \times (1\ mod\ 13) \times 3\ mod\ 13 = 3\ mod\ 13$.

E1.9 - 4 Find $x = 7^{100}\ mod\ 13$
Since $100 > 13$ We use **Euler's Theorem.**
$\phi(13) = 12$**,** cause 13 is prime and so $\phi(p) = p - 1$.
$100 = 7 \times 12 + 4$ and $7^{100} = (7^7)^{12} \times 7^4$
Since $7^{\phi(13)} = 1\ mod\ 13$, we have that $7^{100} mod\ 13 = 1^7\ mod\ 13 \times 7^4 mod\ 13 =$
$2401\ mod\ 13 = 9\ mod\ 13$.

E1.9 - 5 Find x when $7^x = 11\ mod\ 13$
Go by trials, because this is a DLP and so it is hard to solve.
When we have a small modulo, we can go on finding that for $x = 5$, we have that $7^5 mod\ 13 = 11\ mod\ 13$.

2015 Jan 14th

Q5.1 Φ(7) = ? (Φ is the Euler's totient function)

Must find how many elements are coprime with 7, i.e. gcd(a, m) = 1 for each element a in $Z_m = \{0, 1, 2, 3, 4, 5, 6\}$.

We notice that, since $m = 7 = p$ is prime, $\phi(7) = p - 1 = 6$.

Q5.2 $\phi(12) = ?$

$Z_m = Z_{12} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$. Must find all elements that are coprime with 12, i.e. the multiplicative group Z^*_{12}= {1, 5, 7, 11}, so that $\phi(12) = 4$.

Q5.3 Find the multiplicative inverse of 5 (mod 6).

An inverse exists, since 5 and 6 are coprime. It is like to ask to find the Multiplicative Inverse of 5 in Z_6. In modular arithmetic mod n, the Multiplicative Inverse of x is the integer y such that $(x \times y)\, mod(n) = 1\, mod(n)$, thus we have to find that number y such that (5*y)mod(6) = 1mod(6).

In this case, y=5 itself, cause 25mod(6) = 1mod(6).

2015 Feb 10th, 2015 March 26th

Q6.1 $\phi(10) = ?$

$Z_m = Z_{10} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Must find all elements that are coprime with 10, i.e. the multiplicative group $Z^*_{10} = \{1, 3, 7, 9\}$, so that $\phi(10) = 4$.

Q6.3 What is the multiplicative group Z^*_{10}?

The multiplicative group Z^*_m is the set of positive numbers in Z_m that are relatively coprime with m. So, $Z^*_{10} = \{1, 3, 7, 9\}$.

2015 Nov 4th

Q5.3 Determine the multiplicative inverse of 47 mod 64

An inverse exists, since gcd(64, 47)=1 (so they are coprime). We have to find y such that (47*y)mod(64) = 1mod(64). Here is a step by step solution using the Extended Euclidean Algorithm.

$$r_0 = 64, r_1 = 47$$

1. $64 = 1 \times 47 + 17$ $\quad r_0 = q_1 \times r_1 + r_2$, with $r_2 = s_2 \times r_0 + t_2 \times r_1$
2. $47 = 2 \times 17 + 13$ $\quad r_1 = q_2 \times r_2 + r_3$, with $r_3 = s_3 \times r_0 + t_3 \times r_1$

Repeat the iterations following the rule:

$r_{l-2} = q_{l-1} \times r_{l-1} + r_l$ with $r_l = s_l \times r_0 + t_l \times r_1$

3. $17 = 1 \times 13 + 4$ $\quad r_2 = q_3 \times r_3 + r_4$ with $r_4 = s_4 \times r_0 + t_4 \times r_1$
4. $13 = 1 \times 4 + 1$ $\quad r_3 = q_4 \times r_4 + r_5$ with $r_5 = s_5 \times r_0 + t_5 \times r_1$

Stop when $r_l = 1$ in order to satisfy the Bézout's identity s.t. $s \times r_0 + t \times r_1 = gcd(r_0, r_1) = 1$, so r_0, r_1 are coprime. In our case, $r_l = r_5$

5. $1 = 13 - 3 \times 4$ $\quad r_5 = r_3 - q_4 \times r_4$ (line 4)

Since $r_4 = r_2 - q_3 \times r_3$(line 3) we have that $4 = 17 - (1 \times 13) = 17 - 13$, so write:

6. $1 = 13 - 3 \times 4 = 13 - 3 \times (17 - 13) = 4 \times 13 - 2 \times 17.$

However this result is not in the form $1 = s_l \times r_0 + t_l \times r_1$ and thus we need to **shape our result**. A suggestion could be to use $r_0$and $r_1$inside the operations.
We would like to have values such that $r_0 = 64, r_1 = 47.$

7. $= 4 \times 13 - 2 \times 17 = 4 \times (47 - 2 \times 17) - 3 \times 17 =$
$= 4 \times 47 - 11 \times 17 = 4 \times 47 - 11 \times (64 - 47) = 15 \times 47 - 11 \times 64.$

Now we have obtained the desired form and we pick that t that multiplies $r_1 = 47$.
So, the multiplicative inverse of 47 mod 64 is 15.

Q5.4 Given the two primes 23 and 11 find integer α such that $\alpha^{11} = 1 \, mod \, 23$
Must apply the **Euler's Theorem**: first of all, α and $m = 23$ should be coprime, s.t. $gcd(\alpha, 23) = 1$. Standing to the theorem, since $m = 23$ is prime, anything elevated to $\phi(m) = 22$is equal to $1 \, mod \, m = 1 \, mod \, 23$: we have to find an α s.t. $\alpha^{11} = h^{22}$, with h = any number.
Let $q = 11$.
Chose a number h such that $1 < h < m - 1$. e.g. $h = 2$.
Compute $g = h^{\frac{m-1}{q}} = 2^{\frac{23-1}{11}} = 4$.
If we assign the value of g to α then we'll have that $4^{11} = 1 \, mod \, 23$.
Why pick $4 = 2^2$? Because $(2^2)^{11} = 2^{22}$, and this is exactly of the form $\alpha^{\phi(m)}$.
Any other number β^k s.t. $k \times 11 = 22$ $[k \times q = \phi(m)]$ would be good.

2016 Feb 12th
Q6.1 Compute $5^{12241} \, mod \, 13$
Use examples in the **Euler's Theorem.**
$\phi(13) = 12$: since 13 is prime, $\phi(p) = p - 1$.

$$12241 = 12 \times 1020 + 1$$
$$5^{12241} = (5^{12})^{1020} \times 5$$

Since $5^{\phi(13)} = 1 \, mod \, 13$, we have that $5^{12241} = 1^{1020} \times 5 \, mod \, 13 = 5 \, mod \, 13$.

2017 Jul 24th
Q5.2 Find the multiplicative inverse of 19 mod 37
A multiplicative inverse exists since $gcd(37, 19) = 1$, so the numbers are coprime.
Using the Bézout's identity we have to find that r_l such that $s \times r_0 + t \times r_1 = gcd(r_0, r_1) = 1$.

$$r_0 = 37, r_1 = 19$$

1. $37 = 1 \times 19 + 18$ $\quad r_0 = q_1 \times r_1 + r2$ with $r_2 = s_2 \times r_0 + t_2 \times r_1$
2. $19 = 1 \times 18 + 1$ $\quad r_1 = q_2 \times r_2 + r_3$ with $r_3 = s_3 \times r_0 + t_3 \times r_1$

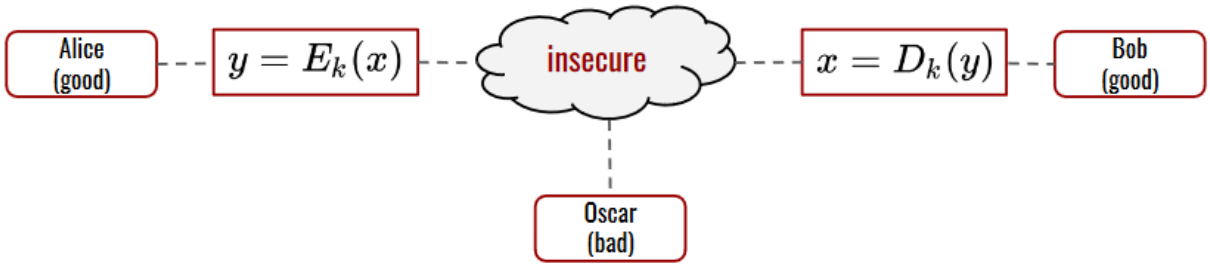
Stop! Since $r_3 = 1$. Now we want to find the value t that multiplies r_1, so we need to reshape our solution.

3. $1 = 19 - 1 \times 18$ (line 2)
4. $19 - 1 \times (18) = 19 - 1 \times (37 - 1 \times 19)$ (line 1)
5. $= 2 \times 19 - 37 = 1.$

This result is in the shape we want, i.e. $s \times r_0 \quad + t \times r_1 = 1.$

We want that t that multiplies r_1: the multiplicative inverse of 19 in Z^*_{37} is so $t_3 = 2$.

Symmetric Cryptography



Q: why do we need a key? Cannot just we use E and D?

Remark. We assume that the symmetric key k is exchanged through a secure channel (see protocols for key exchange)

In Symmetric Ciphers we use the same key k both for encryption and decryption.
Some examples of Symmetric Ciphers are the **Caesar's cipher** and **substitution cipher**, that can be easily broken by brute force or letter frequency analysis.

Perfect Cipher

Given a plaintext space = $\{0, 1\}^n$, D known, and a ciphertext y then the probability that exists a key k such that $D_k(y) = x$ for any plaintext x is equal to the apriori probability that x is the plaintext:

$$Pr[x|y] = Pr[x]$$

In other words, the ciphertext does not reveal any information on the plaintext.

$$Pr[x|y] = Pr[x \wedge y] \,/\, Pr[y] \qquad \text{(conditional probability)}$$
$$Pr[x \wedge y] = Pr[x|y]\, Pr[y] = Pr[y|x]\, Pr[x] \qquad \text{(Bayes)}$$
$$Pr[x \wedge y] = Pr[x]\, Pr[y] \qquad \text{(if x and y are independent)}$$

Then:

$$Pr[x]\, Pr[y] = Pr[y|x]\, Pr[x] \Rightarrow Pr[y] = Pr[y|x]$$

One Time Pad (OTP)

The **One Time Pad (OTP)** is a perfect cipher, but a True Random Number Generator (TRNG) is needed.

- Plaintext space: $\{0,1\}^n$
- Key space: $\{0,1\}^n$
- Symmetric scheme, key chosen using a True Random Number Generator (TRNG)
- key is only used once (i.e., two messages must be encrypted with two different keys)

$$y = E_k(x) = x \oplus k$$

$$x = D_k(y) = y \oplus k = (x \oplus k) \oplus k = x \oplus (k \oplus k) = x \oplus 0$$

Shannon's Theorem

> Shannon's Theorem: A cipher cannot be perfect if the size of its key space is less than the size of its message space.

Proof by contradiction.

$$2^{|k|} < 2^{|x|}$$

$Pr[y_0] > 0$ (ciphertext must exists)

$S = \{D_k(y_0) : k \in K\}$ (K is the set of all possible keys)

Then: $\exists\, x$ such that $x \notin S$

If we know: $\forall\, k \in K : E_k(x) \neq y_0 \Rightarrow Pr[y_0] = 0$

Symmetric Cryptography can be done in two main approaches:

- **Stream Ciphers**: given a secret key, generate a byte of stream called keystream that has the same length as the message. Encryption and decryption are done by XORing individual bits of the keystream and the message. OTP is an example. **PROS** easy to implement. **CONS** a key of the same length of the message is needed.
- **Block Ciphers:** split the message in x blocks of fixed size, encrypting and decrypting each block. Various 'operational modes' exist. **PROS** a key with a shorter size than the message length can be used **CONS** tricky to implement and must be aware of some mathematical vulnerabilities.

Stream Ciphers

Given plaintext x (x_i is the i-th bit from x), keystream s (where |x|==|s|, s_i is the i-th bit from s):

- Encryption: $y_i = E_{s_i} = x_i \oplus s_i = x_i + s_i \bmod 2$
- Decryption: $x_i = D_{s_i} = y_i \oplus s_i = y_i + s_i \bmod 2$

A stream cipher is called:

- synchronous when s_i is a function of the key
- asynchronous when is s_i a function of the key and previous bits of y

The cipher must provide the keystream generator.
Notice that modulo 2 is equivalent to a bitwise XOR operation.

In stream ciphers the idea is to **simulate the ONE TIME PAD**, and so the key, called **seed**, is used to generate a byte stream called **keystream**.

Some well known Stream Ciphers are: A5/1 (**GSM**) and Linear Feedback Shift Registers (LFSR), **RC-4** (Ron's Code or Rivest Cipher 4).

RC-4 Ron's Code (or Rivest Cipher 4)

It is a **stream cipher** designed in 1987 by Ron Rivest.
It is **synchronous,** with a **variable key length** and it is very fast to compute: starting from the key it will generate an apparently random permutation, eventually it will repeat the sequence. However, the period for repetition is very long (> 10^100).

Since we cannot use the same key for two messages, we need a way to randomize each keystream, maybe by using an Initialization Vector **(IV)** acting as randomizer.
As with any stream cipher, these can be used for encryption by combining it with the plaintext using bitwise exclusive-or; decryption is performed the same way (since exclusive-or with given data is an involution). This is **similar to the one-time pad** except that generated *pseudo random bits*, rather than a prepared stream, are used.

This algorithm is used for **WEP (Wired Equivalent Privacy),** where IV is 24 bits and the key is 40 bits. However, there is a 50% prob that the same IV will repeat after 5000 packets.

Several attacks along the years. From 2015, there is speculation that some state cryptologic agencies may possess the capability to break RC4 when used in the TLS protocol. IETF has published RFC 7465 to **prohibit the use of RC4** in TLS; Mozilla and Microsoft have issued similar recommendations.

RC-4 KEY SCHEDULING ALGORITHM (KSA)

S is an array of 256 integers.

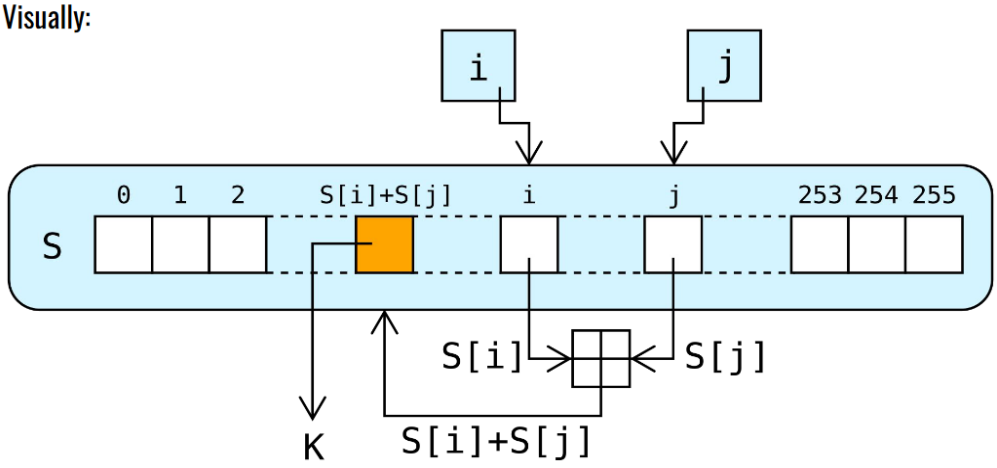
```
int j = 0;
for(int i = 0; i < 256; i++)
    S[i] = i;
for(int i = 0; i < 256; i++) {
    j = (j + S[i] + key[i % len]) % 256;
    swap(&S[i], &S[j]);
}
```

At the end S is a permutation of {0, ..., 255} generated based on the value of the key.

RC-4 PSEUDO-RANDOM GENERATION ALGORITHM (PRGA)

```
int i = 0, j = 0;
for(size_t n = 0, len = strlen(plaintext); n < len; n++) {
   i = (i + 1) % 256;
   j = (j + S[i]) % 256;
   swap(&S[i], &S[j]);
   int K = S[(S[i] + S[j]) % 256];   // keystream byte
   ciphertext[n] = K ^ plaintext[n]; // XOR encryption
}
```

Visually:



So in **RC-4** at every iteration we compute the **XOR** between k and the next byte of the plaintext (or ciphertext), making this a **synchronous stream cipher**. The algorithm is used in order to generate the key to encrypt the next byte of the plaintext.

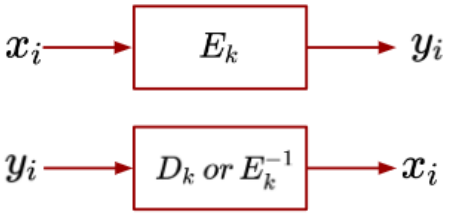
To recap, we can say that RC4 is a **synchronous stream cipher**.
The algorithm generates a pseudorandom stream of bits called **keystream**, which is combined with the plaintext thanks to a bitwise XOR operation.
To generate the keystream it uses a **permutation S of the first 256 natural numbers** and two index pointers i and j. To generate the permutation a key of variable length is used, with the **key scheduling algorithm (KSA)**.
For every i in the array, j gets the value $(j + S[i] + k[i])\ mod\ 256$, then $S[i]$ and $S[j]$ are **swapped**.
Now the **pseudo random generation algorithm (PRGA)** is used. The two indices i, j are set to 0, then as long as necessary i is incremented to $(i + 1)\ mod\ 256$, then j becomes $(j + S[i])\ mod\ 256$.
$S[i]$ and $S[j]$ are swapped and the value K which corresponds to $S[\ (S[i] + S[j])\ mod\ 256]$is returned as output and it is XORed with the next byte of the plaintext (or ciphertext).

Block Ciphers

Given:

- a block x_i of the message x of h bits (h is fixed)
- a key k of n bits (n fixed)
- n can be different from h

Then:

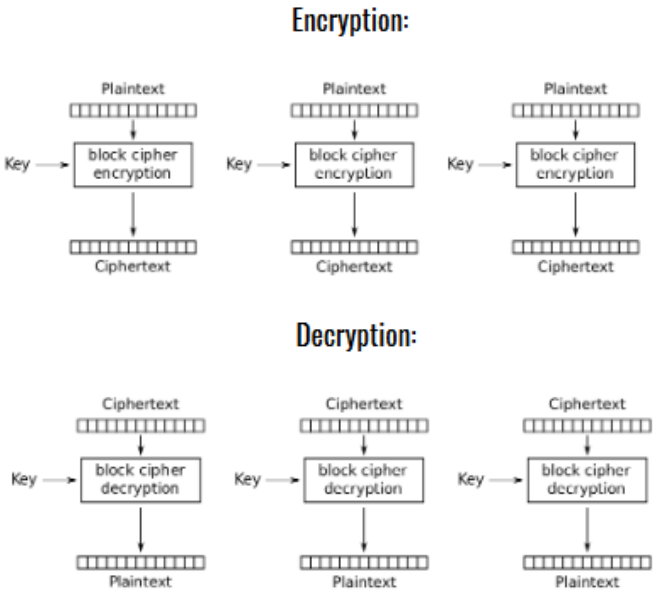


Real world block ciphers are: DES, 2-DES, **3-DES**, IDEA, RC-2, RC-5, Blowfish, **AES**.

Operational Modes

What if the message is larger than the block size? It is possible to use some **Operational Modes ,** such as Electronic Code Block (**ECB**), Cipher Block Chaining (**CBC**), Propagating Cipher Block Chaining (**PCBC**), Output FeedBack Mode (**OFB**), Cipher Feedback Mode (**CFB**), Counter Mode (**CTR**).

Electronic Code Block (ECB)

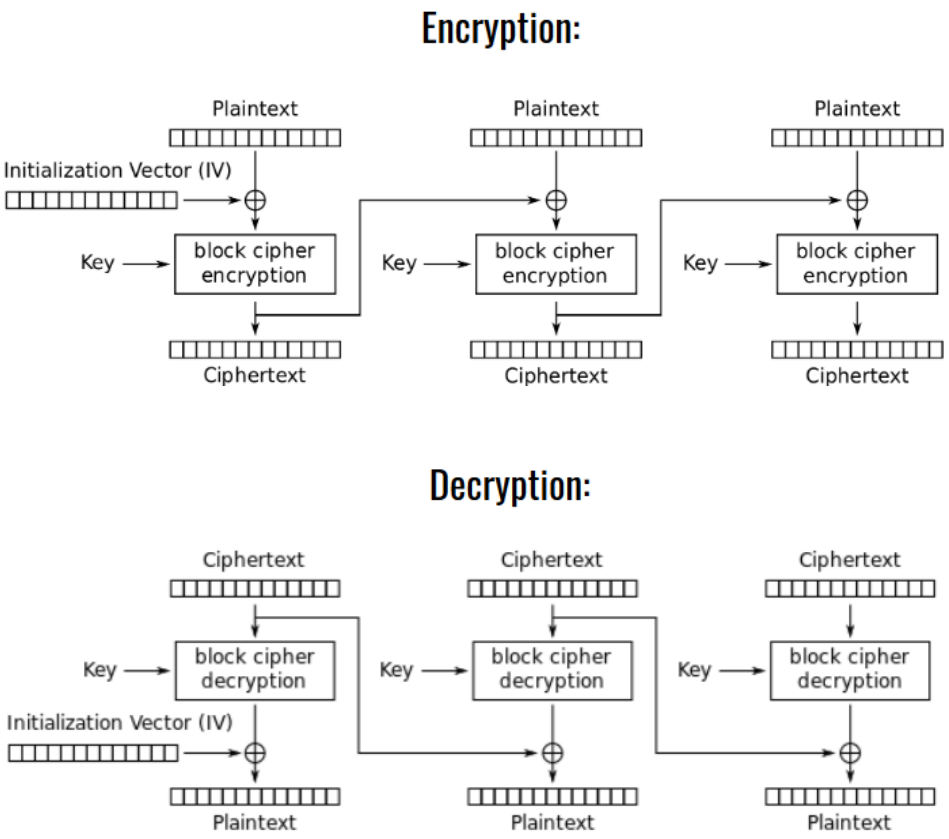


Encryption and Decryption can be easily **parallelized**.

Some **attacks** are possible: blocks can be reordered, replaced, removed or appended.

Encrypting the same message **twice** will generate the same ciphertext.

Cipher Block Chaining (CBC)



Parallelization: encryption is not parallelizable, while decryption is.
Block y_i depends on y_{i-1}.
If one ciphertext bit x_i is flipped, then all subsequent blocks are affected.
If one bit is flipped in a block y_{i-1}, then $y_i$is affected in an unpredictable manner, while x_i is affected in a predictable manner. This could be exploited by an attacker.

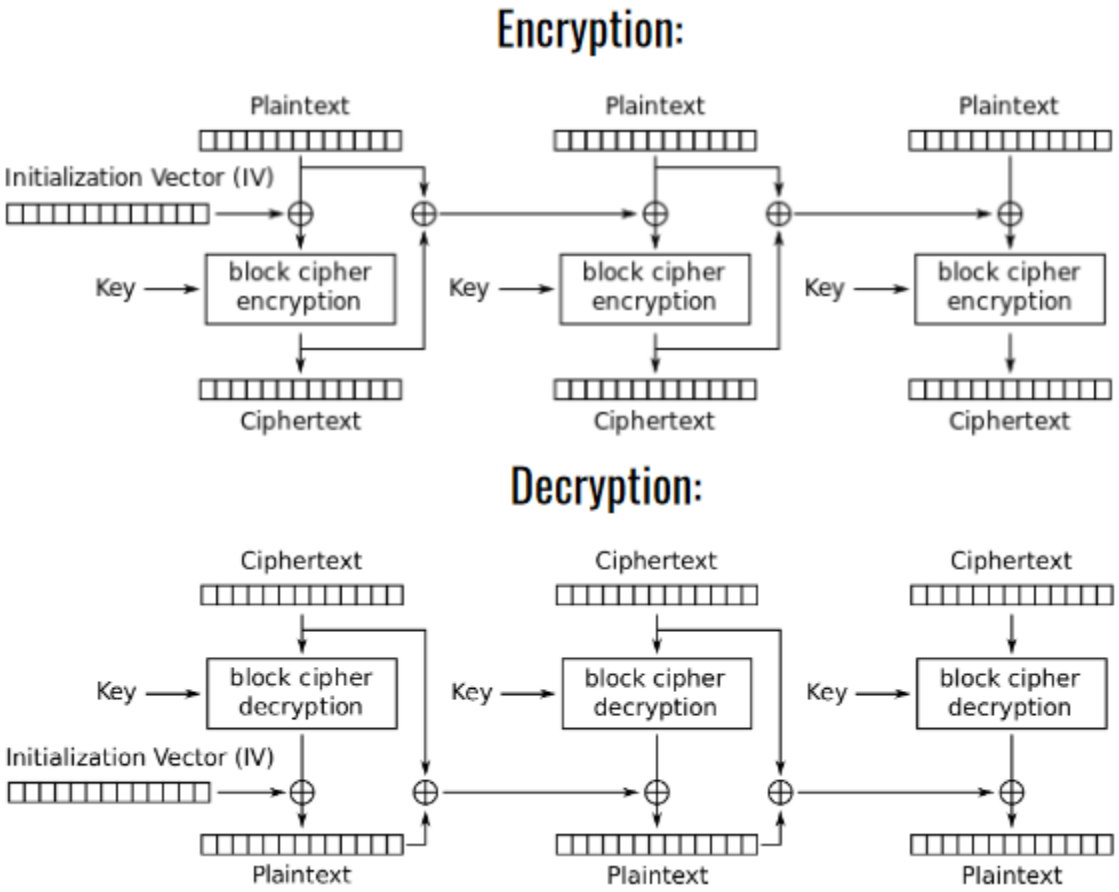
The **Initialization Vector (IV)** should be different for each message, otherwise an attacker can understand when we are encrypting again the same message. It **can be public**.

CBC can be seen as an asynchronous stream cipher.

The message **must be padded** to a size multiple of the block size, because a block cipher can deal only with a fixed block size.
Padding can be done by **adding information**, but however this technique increases the size of the text, or using the **ciphertext stealing** technique, that consists, **in encryption**, in swapping the last cipher blocks, then truncate the ciphertext to the original length of the plaintext and, **in**

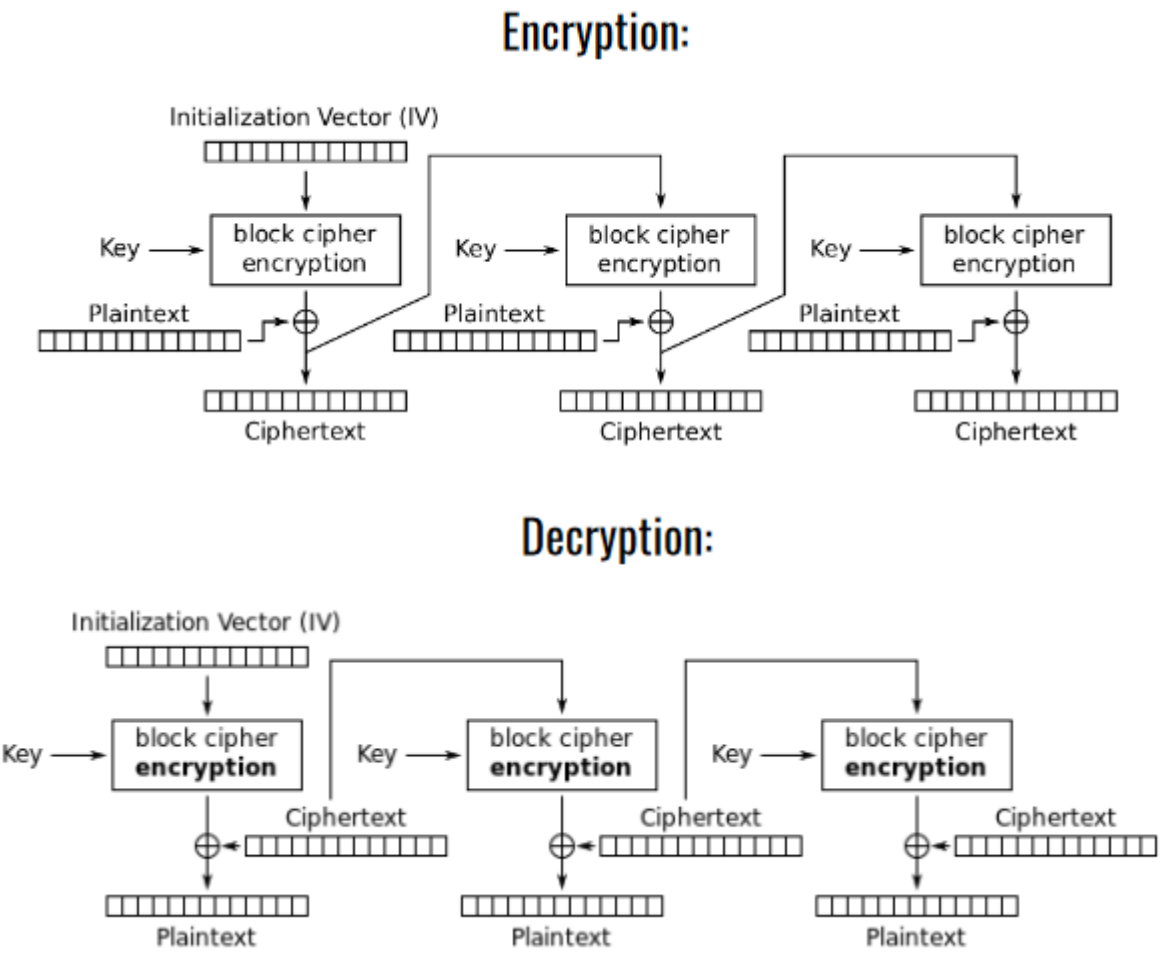
decryption, in swapping the last cipher blocks, decrypt, then truncate the plaintext to the original length of the ciphertext.

Propagating Cipher Block Chaining (PCBC)



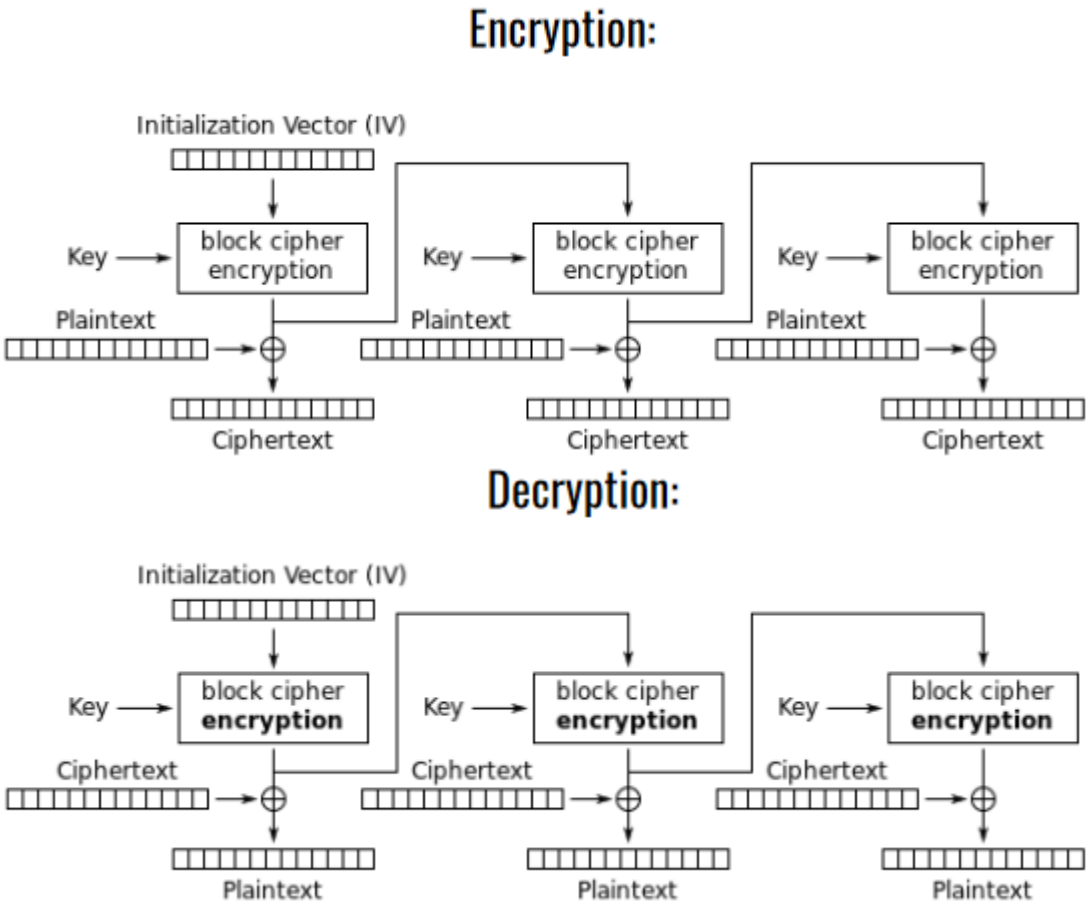
It is designed to propagate small changes to all subsequent blocks both during encryption and decryption. However, if two adjacent ciphertext blocks are exchanged, subsequent decrypted blocks are not affected.

Cipher Feedback (CFB)



Can be viewed as an **asynchronous stream cipher**, because the encryption of a new block **depends from the ciphertext of the previous block**, that is the XOR between the plaintext and a keystream s.
Uses an **IV:** at every block, a portion of plaintext is XORed with the IV, and this produces that portion of ciphertext that is used again as IV for the next block.
One bit flip in ciphertext only affects one bit in the output: using error correction codes may be helpful.
Parallelization: encryption is not parallelizable, while decryption is.
The same encryption algorithm is both used for encryption and decryption.

Output FeedBack (OFB)

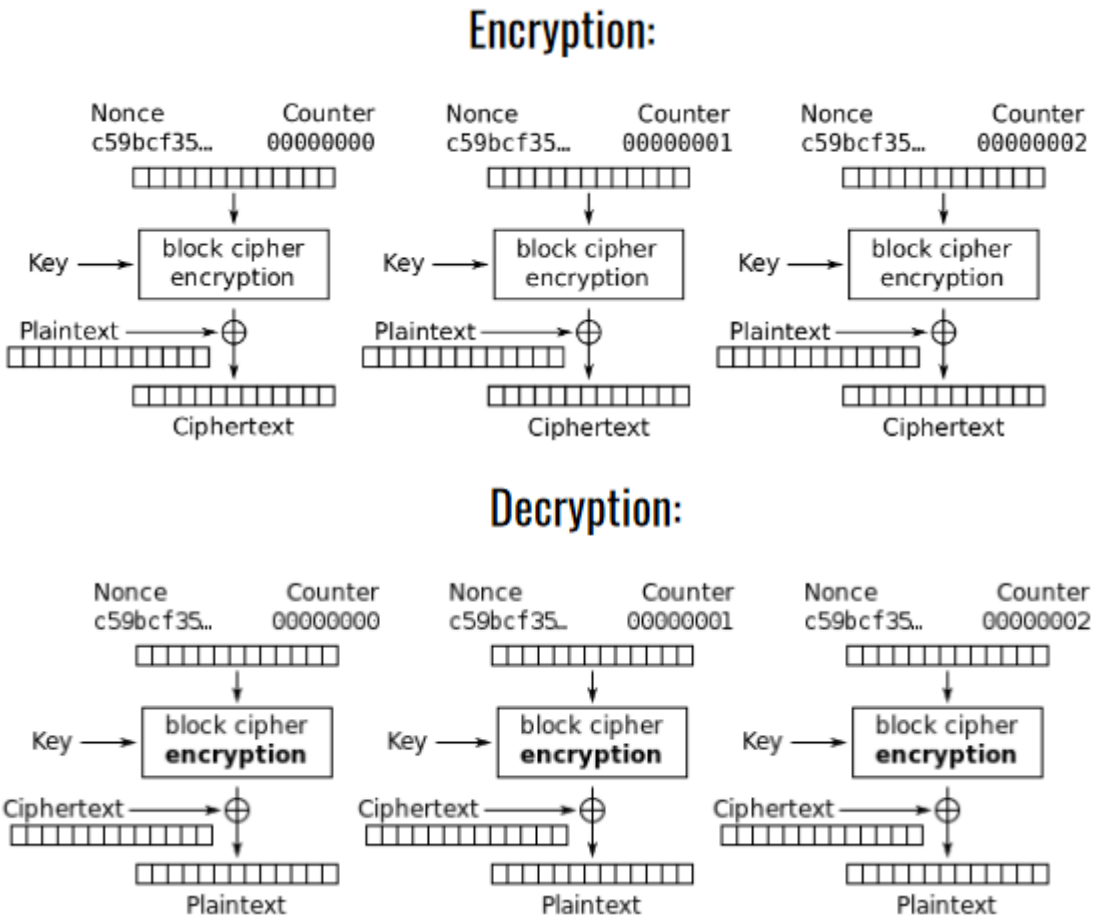


It is very similar to the CFB except that now the input for the next block is taken **before** the XOR operation.
Differently from CFB, OFB can be viewed as a **synchronous stream cipher**, because the input of the new block only depends from a keystream s that is not XORed with the plaintext: a block is **not influenced by the previous ciphertext**.
As CFB, one bit flip in ciphertext only affects one bit in the output: using error correction codes may be helpful.
Parallelization: neither encryption nor decryption are parallelizable.
The same encryption algorithm is both used for encryption and decryption.

Problems: if the encryption function and the key are known to the adversary, then IV must be kept secret; if the key is not known by the attacker, then IV can be sent in clear; IV must be different for each message.

It is used for stream-oriented **transmissions over noisy channels**, for example on satellite communication.

Counter Mode (CTR)



Besides IV, it uses a counter that is incremented for each block called **nonce**, i.e. '**n**umber that is used only **once'**.
Parallelization: both encryption and decryption are parallelizable.
The same encryption algorithm is both used for encryption and decryption.

Summary - Operation Modes

Mode	Description	Typical Application
Electronic Codebook (ECB)	Each block of 64 plaintext bits is encoded independently using the same key.	• Secure transmission of single values (e.g., an encryption key)
Cipher Block Chaining (CBC)	The input to the encryption algorithm is the XOR of the next 64 bits of plaintext and the preceding 64 bits of ciphertext.	• General-purpose block-oriented transmission • Authentication
Cipher Feedback (CFB)	Input is processed *s* bits at a time. Preceding ciphertext is used as input to the encryption algorithm to produce pseudorandom output, which is XORed with plaintext to produce next unit of ciphertext.	• General-purpose stream-oriented transmission • Authentication
Output Feedback (OFB)	Similar to CFB, except that the input to the encryption algorithm is the preceding encryption output, and full blocks are used.	• Stream-oriented transmission over noisy channel (e.g., satellite communication)
Counter (CTR)	Each block of plaintext is XORed with an encrypted counter. The counter is incremented for each subsequent block.	• General-purpose block-oriented transmission • Useful for high-speed requirements

	ECB	CBC	CFB	OFB	CTR
	Electronic Code Book	Cipher Block Chaining	Output Feedback	Cipher Feedback	Counter
Information leakage	High	low	low	low	low
Encryption parallelizable	Yes	No	No	No	Yes
Decryption parallelizable	Yes	Yes	Yes	No	Yes
Ciphertext manipulation	Yes	No	No	No	No
Precompute	No	No	No	Yes	Yes
Encryption error propagation	No	Yes	Yes	No	No
Decryption error propagation	No	Partial (2 Blocks)	Partial (2 Blocks)	No	No

Shannon suggested two operations for a cipher:

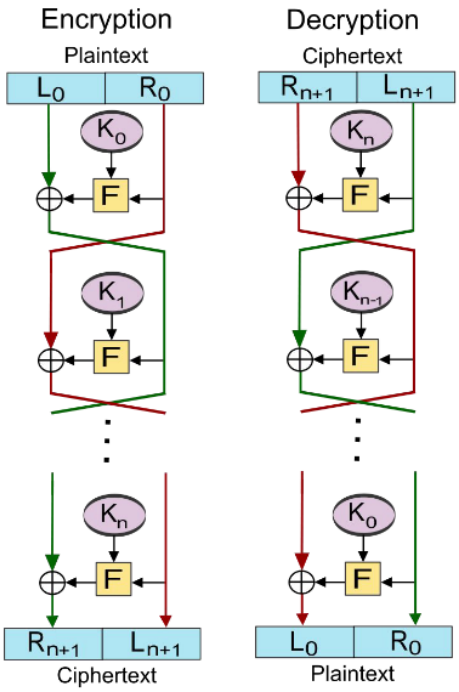
- **Confusion**: each binary digit (bit) of the ciphertext should depend on several parts of the key, obscuring the connections between the two. It hides the relationship between the ciphertext and the key and makes it difficult to find the key from the ciphertext and if a single bit in a key is changed, the calculation of the values of most or all of the bits in the ciphertext will be affected. One way of achieving confusion is substitution (as in AES and DES).
- **Diffusion**: changing a single bit of the plaintext, then (statistically) half of the bits in the ciphertext should change, and similarly, if we change one bit of the ciphertext, then approximately one half of the plaintext bits should change. Since a bit can have only two states, when they are all re-evaluated and changed from one seemingly random position to another, half of the bits will have changed state. The idea is to hide statistical properties of the plaintext. One way of achieving diffusion is bit permutation (done in DES).

DES - Data Encryption Standard

Developed by IBM in 1970, it is a symmetric key cipher based on Feistel Network.
With a **key length of 56 bits**, it is now considered insecure and it can be broken in less than 24 hours using a “linear attack”, that is why it has been replaced with AES.
It is a **block cipher** and uses blocks of 64-bit size.

Feistel Network

- This design allows encryption and decryption to be the same/similar. Hence, function F does not have to be invertible.
- It has been proved that if F is pseudorandom function with K_i used as seeds (subkeys derived from the secret key k) then sufficient to make it a "strong" pseudorandom permutation.
- DES is based on this design. We do see not its internal design (F and key derivation), see slides of chapter 3 of Book “Understanding Cryptography” for more details.



DES can be broken: how to cope with this fact? Two main approaches are proposed:

1. Move to another cipher, e.g. AES
2. “Fix” DES with different strategies

3-DES : this strategy consists in iterating the encryption of the plaintext applying DES for three times with three different keys **(EEE mode)**. $y = E_{k1}(E_{k2}(E_{k3}(x)))$.

EDE mode : only requires two keys. $y = E_{k1}(D_{k2}(E_{k1}(x)))$.

2-DES with one key : $y = E_{k1}(E_{k1}(x))$. The brute force complexity is not changed: still requires 2^{56} attempts. The only difference is that each attempt costs twice as one attempt for DES.

2-DES with two keys

$$y = E_{k_2}\left(E_{k_1}\left(x\right)\right)$$
$$x = D_{k_1}\left(D_{k_2}\left(y\right)\right)$$

Q. Is the attack complexity 2 times 2^56 since we use two keys of 56 bits each?
A. NO, effective key strength is only 2^57 and not 2^112.

Meet-in-the-Middle Attack. Assuming to have a pair (x, y):

- 2^56 attempts: $A = \{\forall\, k_1 : E_{k_1}\left(x\right)\}$
- 2^56 attempts: $B = \{\forall\, k_2 : D_{k_2}\left(y\right)\}$
- Find matching: (a, b) such that $a = b$ where $a \in A, b \in B$
- first key is the one that have generated a, second one is the one that generated b
- Total attempts: 2^56+2^56=2^57

Remark: Using MITM, we can see that 3-DES thus has 2^112 key strength

DES-X - KEY WHITENING : using a XOR-encrypt-XOR form, using a simple XOR with the key before the first round and after the last round of encryption.

DES-X: $\text{DES-X}\left(x\right) = k_2 \oplus \left(\text{DES}_k\left(x \oplus k_1\right)\right)$

- Three keys (k 56 bits, k_1 64 bits, k_2 64 bits): 184 bits
- However, effective key size is only 119 bits when the attacker can obtain enough (plaintext, ciphertext) pairs [paper]

[Paper]

AES - Advanced Encryption Standard

AES - Advanced Encryption Standard a symmetric block cipher chosen by the US NAtional Institute of Standards and Technology (NIST) sinceNovember 2001.
As a block cipher, AES supports 128-bit block size and can operate with three different key lengths: 128, 192 and 256 bit.
Here examples are done with **AES-128**.

Structure Overview

Being a block cipher, AES repeats the encryption (and decryption) operations **for each block of 16 Bytes** (128 bits), that can be seen as a **4x4 matrix of bytes** in the implementation phase: AES is in fact a **byte-oriented cipher**, and this makes it very malleable in terms of data processing.

This matrix is copied into the **State** array, which is modified in each stage of encryption or decryption. After the final stage, State is copied into an output matrix.

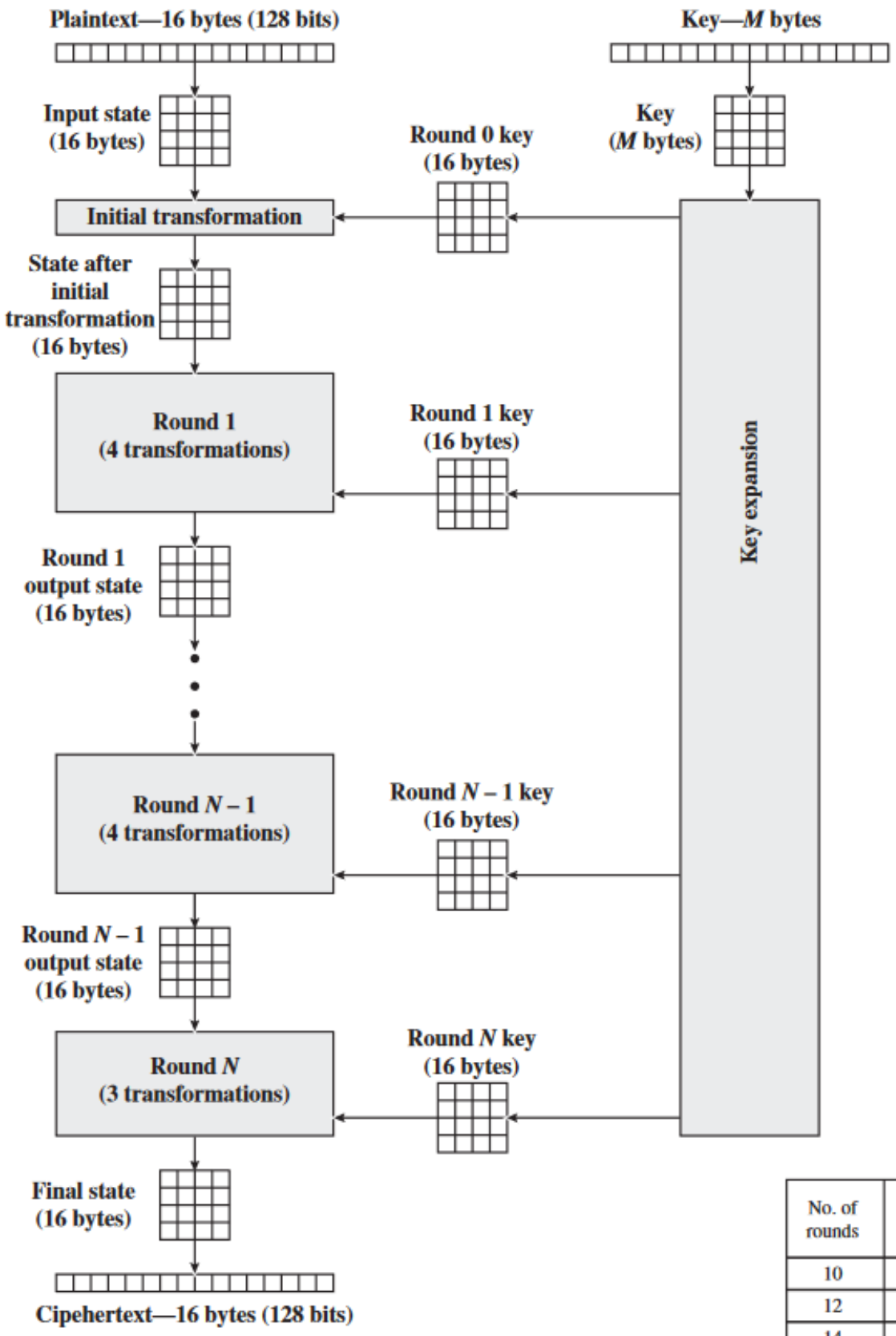
Similarly, the **key** is depicted as a square matrix of bytes. This **key is then expanded** into an array of key schedule words.

More specifically talking about encryption - but the same can be applied in reverse with decryption - for each block of plaintext AES performs a number n_r of rounds, and **each round is constituted by a sequence of layers** in order to satisfy the fundamental properties for a secure cipher identified by Claude Shannon in his 1945 classified report *A Mathematical Theory of Cryptography* such as **Confusion**, **Diffusion** and Key **Whitening**.

The number n_r of rounds is fixed in respect to the chosen key length:

Key length (bits)	Number of rounds
128	10
192	12
256	14

The first $n_r - 1$ rounds consist of four different transformation functions: **SubBytes**, **ShiftRows**, **MixColumns** and **AddRoundKey**. The final round only contains three transformations and there is an initial single transformation from AddRoundKey before the first round.

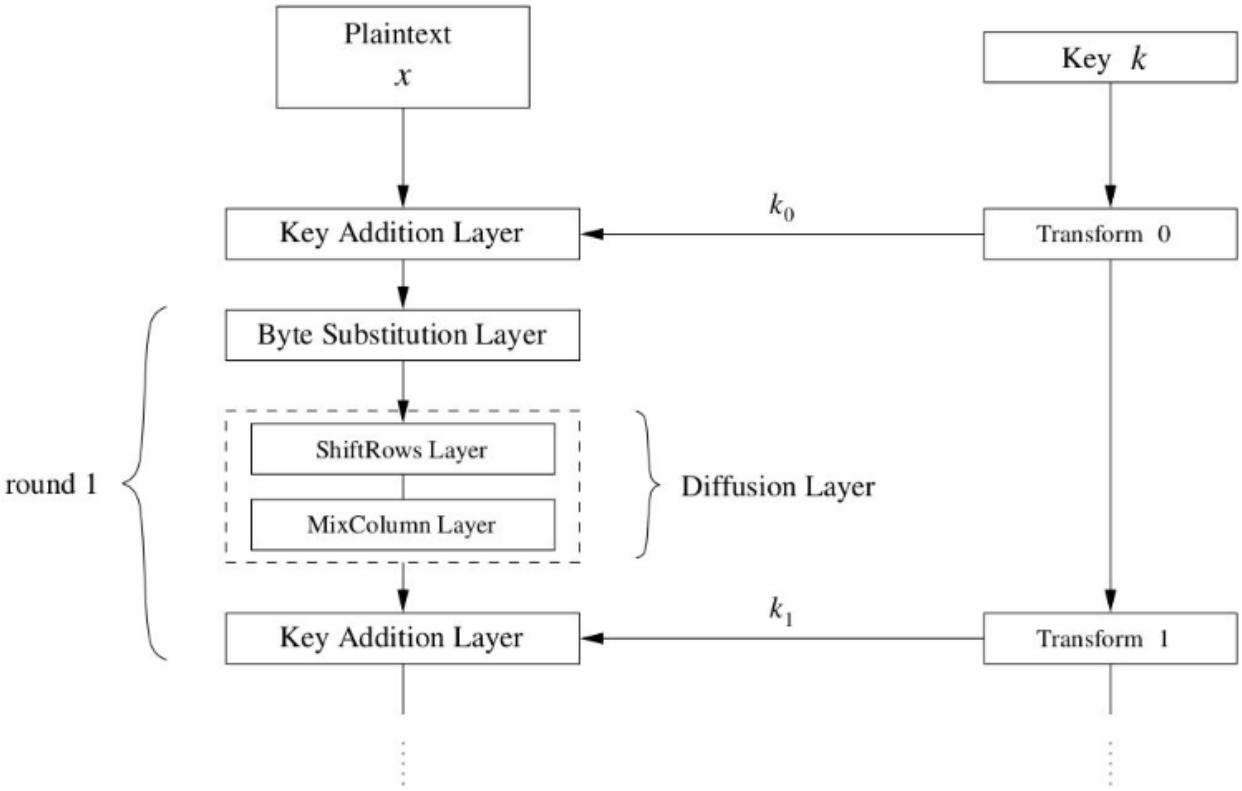


No. of rounds	Key Length (bytes)
10	16
12	24
14	32

Figure 5.1 AES Encryption Process

- **Byte Substitution** Layer is for Confusion
- **Shift Rows** and **Mix Column** Layers are for Diffusion
- **Key Addition** Layer is for Key Whitening.

In each Round those layers are repeated (except for the last round that misses the Mix Columns Layer).



Chapter 4 of Understanding Cryptography by Christof Paar and Jan Pelzl

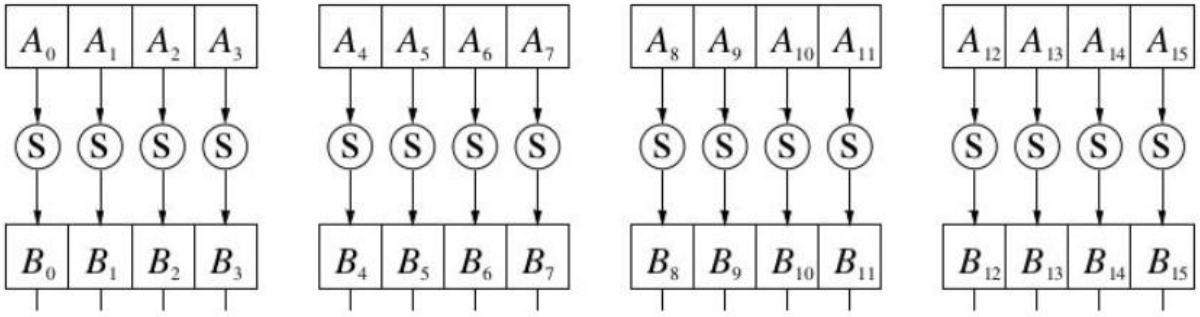
Input and State

As a block cipher, AES reads 16 bytes (128 bits) per time and puts them into a matrix called **State**. If the last block has less than 16 bytes, then a padding technique can be used (e.g. the **PKCS7** padding approach). With the state it is also taken in input the key, but this will be discussed later.
Here, $A_0, \ldots, A_{15}$ denote the 16-byte input of AES.

A_0	A_4	A_8	A_{12}
A_1	A_5	A_9	A_{13}
A_2	A_6	A_{10}	A_{14}
A_3	A_7	A_{11}	A_{15}

Byte Substitution Layer - Confusion

Consists in literally substituting each byte A_i of the State with the correspondent value B_i in an **S-Box**. An S-Box is a lookup table.
Each byte A_i is seen as an element in GF(256). In the first step it is computed the multiplicative inverse in GF(256), then the output is again obfuscated with XOR operations with a fixed polynomial, such that $S(A_i) = B_i$.



x \ y	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

S-Box and Inverted S-Box can be precomputed (some are easily findable on the internet) and stored in appropriate arrays when writing the code.

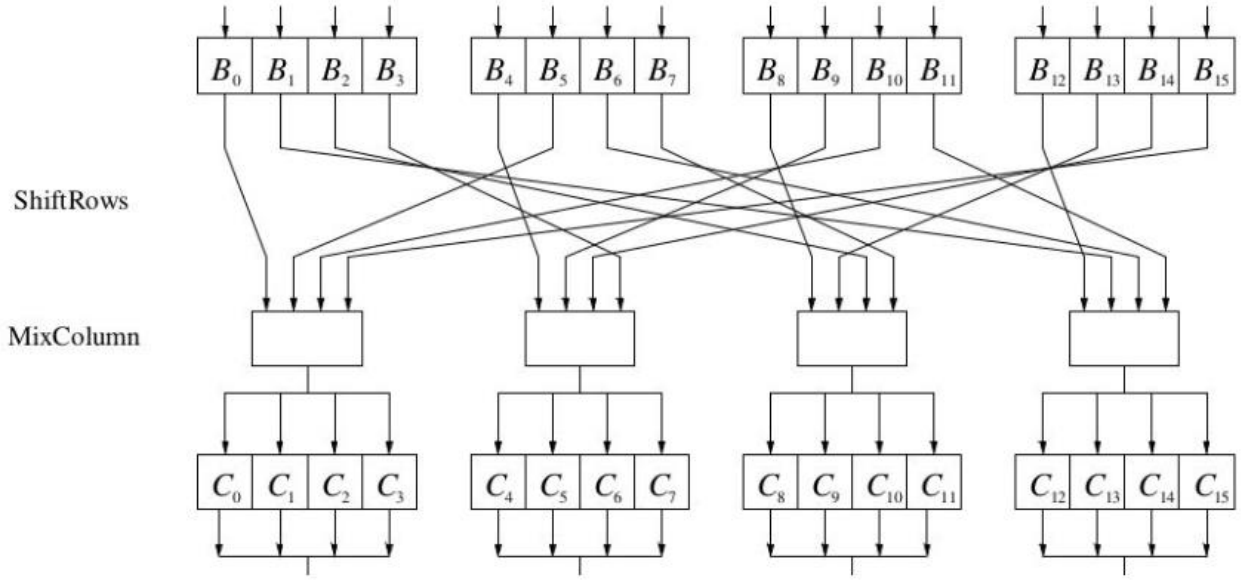
This is an example of S-Box. Here,

$$S(0xC2) = 0x25$$

At the end of this layer, the input state will be constituted of substituted bytes fromthe original input.

Shift Rows and Mix Column Layers - Diffusion

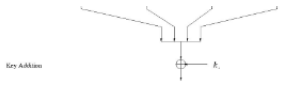
Performs a linear operation on state matrices A, B, i.e DIFFUSION(A) + DIFFUSION(B) = DIFFUSION(A+B).



In the **ShiftRows** sublayer, rows of the state matrix are shifted cyclically:

Input matrix

B_0	B_4	B_8	B_{12}
B_1	B_5	B_9	B_{13}
B_2	B_6	B_{10}	B_{14}
B_3	B_7	B_{11}	B_{15}



Output matrix

B_0	B_4	B_8	B_{12}	no shift
B_5	B_9	B_{13}	B_1	← one position left shift
B_{10}	B_{14}	B_2	B_6	← two positions left shift
B_{15}	B_3	B_7	B_{11}	← three positions left shift

In the **MixColumn** sublayer:

- Linear transformation which mixes each column of the state matrix
- Each 4-byte column is considered as a vector and multiplied by a fixed 4x4 matrix, e.g., the leftmost mix column box is:

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \cdot \begin{pmatrix} B_0 \\ B_5 \\ B_{10} \\ B_{15} \end{pmatrix}$$

  - where 01, 02 and 03 are given in hexadecimal notation
  - each row of the matrix is a shift of the previous row

All arithmetic is done in the Galois Field **GF(256)**. E.g.

$$C_0 = 02 * B_0 + 03 * B_5 + 01 * B_{10} + 01 * B_{15}$$

The values in the matrix can be fixed as the same as in the picture above.

Key Addition Layer

In this layer the key given in input in the cipher is XORed with the outcome of the Diffusion Layer.

However, for security aspects, to avoid a repetition of the XOR opera-tion, it is good to **expand the key**: a 16 Bytes (128 bits) key will be expanded to 44 Bytes, to be given as operand with each round's state.
After the expansion we will have 44 subkeys to use as a round key for each round of encryption of a block. So for n_r rounds, n_r subkeys are added to the text, plus one after the final round.

Thus, for each block of 16 bytes to be encrypted or decrypted, at round i the 16-byte state matrix C has to be xored with the 16-byte subkey k_i, such that the output from round i will be $C\ XOR\ k_i$.

It can be a good idea to **precompute** the expansion of the key and use the relative subkeys as needed: this can be done because the expansion operation does not regards the full input, but only the key itself.

The **key expansion** algorithm takes as input a four-word (16-byte) key and produces a linear array of 44 words (176 bytes), so that for each of the 10 rounds one subkey is provided, plus one after the final round.

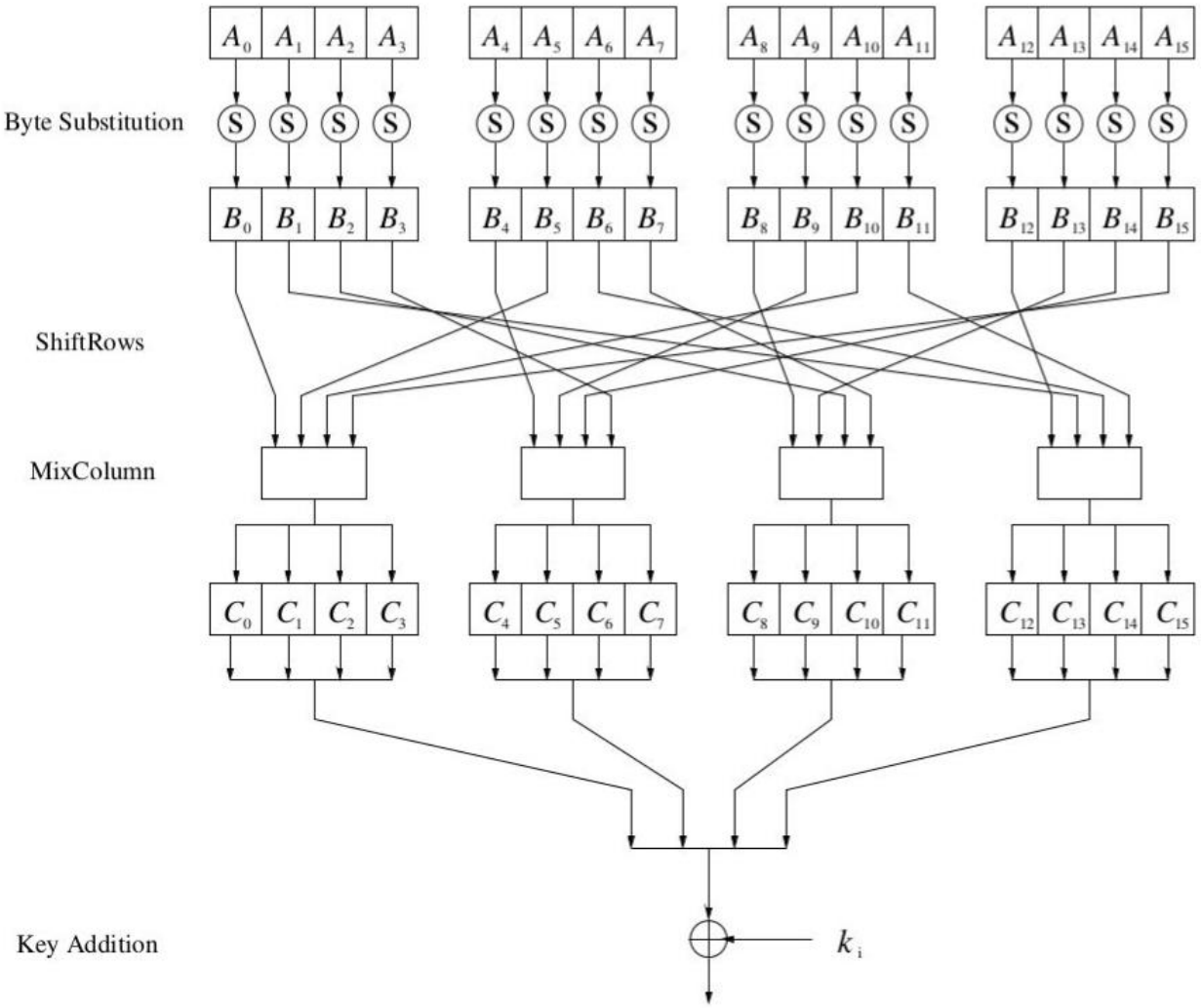
```
KeyExpansion (byte key[16], word w[44])
{
    word temp
    for (i = 0; i < 4; i++)    w[i] = (key[4*i], key[4*i+1],
                                       key[4*i+2],
                                       key[4*i+3]);

    for (i = 4; i < 44; i++)
    {
     temp = w[i - 1];
     if (i mod 4 = 0)    temp = SubWord (RotWord (temp))
                                ⊕ Rcon[i/4];
     w[i] = w[i-4] ⊕ temp
    }
}
```

1. **RotWord** performs a one-byte circular left shift on a word. This means that an input word $[B_0, B_1, B_2, B_3]$ is transformed into $[B_1, B_2, B_3, B_0]$.
2. **SubWord** performs a byte substitution on each byte of its input word, **using the S-Box.**
3. The result of steps 1 and 2 is XORed with a **round constant**, $Rcon[j]$.

The round constant is a word in which the three rightmost bytes are always 0. Thus, the effect of an XOR of a word with Rcon is to only perform an XOR on the left-most byte of the word. The round constant is different for each round and is defined as $Rcon[j] = (RC[j], 0, 0, 0)$, with RC[j] assuming the following values in a table:

j	1	2	3	4	5	6	7	8	9	10
RC[j]	01	02	04	08	10	20	40	80	1B	36

Finally, all the layers will be something like this:



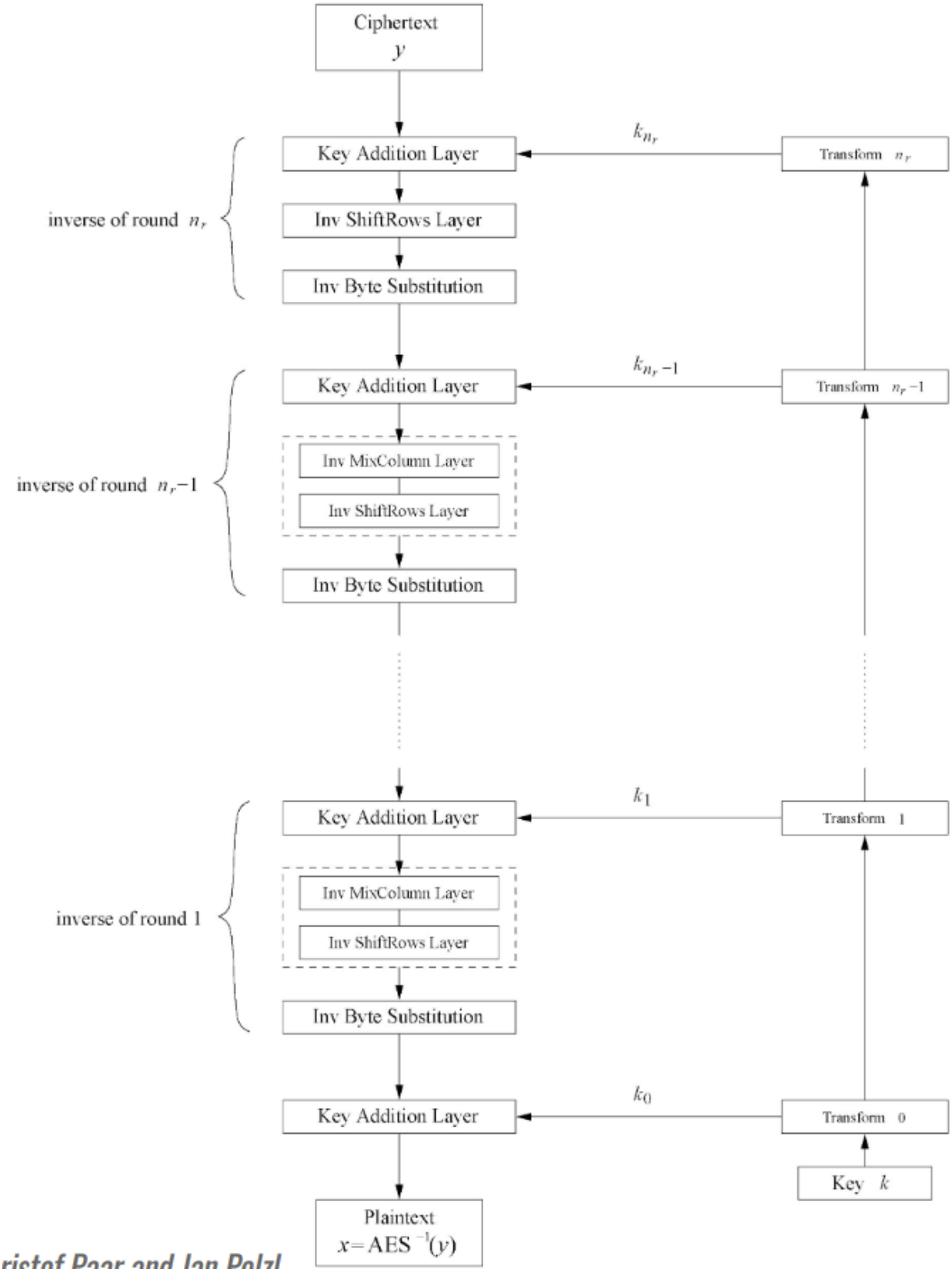
Chapter 4 of Understanding Cryptography by Christof Paar and Jan Pelzl

Decryption

Since AES is not based on a Feistel Network, **all layers must be inverted for decryption**.

- MixColumn -> Inv MixColumn
- ShiftRows -> Inv ShiftRows
- ByteSubs -> Inv ByteSubs

Key addition layer is its own inverse.



ristof Paar and Jan Pelzl

AES and Operation Modes

As a block cipher, AES can be implemented using the operational modes for block ciphers. When needed, the IV must be provided and then a key can be used as the round key.

Examples and exercises

EXERCISE 5.2 Understanding Cryptography
We consider **known-plaintext attacks on block ciphers** by means of an exhaustive key search where **the key is k bits long**. The block length counts n bits with n > k.

1. How many plaintexts and ciphertexts are needed to successfully break a block cipher running in ECB mode? How many steps are done in the worst case?
2. Assume that the initialization vector IV for running the considered block cipher in CBC mode is known. How many plaintexts and ciphertexts are now needed to break the cipher by performing an exhaustive key search? How many steps need now maximally be done? Briefly describe the attack.
3. How many plaintexts and ciphertexts are necessary, if you do not know the IV?
4. Is breaking a block cipher in CBC mode by means of an exhaustive key search considerably more difficult than breaking an ECB mode block cipher?

Remark: in known plaintext, the attacker has samples of both plaintext and ciphertext and is at liberty to make use of them to reveal further secret information such as secret key.

1. Only one block of ciphertext and one of plaintext are needed to break a cipher in ECB mode. In the worst case we have to brute force the key, so with k bits we can generate at most 2^k different keys.
2. In CBC mode, to break the cipher we surely need at least one plaintext block and one ciphertext block. If we can get the first plaintext block, this is enough since we know the IV. However, since for a generic block $i \neq 1$the IV is the precedent ciphertext block, in general for this attack we need one plaintext block i, and two ciphertext blocks i and $i-1$. So **three blocks are needed**. Since the IV is known, we can say that there is not much difference between this situation and an ECB mode, so to find a valid key of k bits we would need to generate 2^k keys.
3. If you do not know the IV then you might try to guess it, so one block of ciphertext and one of plaintext are enough, since the IV (or the previous ciphertext block) has to be generated. To generate the key of $k$bits 2^kattempts are needed, while for the IV (or ciphertext block) of n bits, 2^nattempts are needed. The two operations have to be combinated, so $2^k \times 2^n = 2^{k \times n}$attempts have to be done.
4. If the IV is known, then there is no difference as said in point 2. Else, the asymptotic complexity is similar, too.

EXERCISE 5.3 Understanding Cryptography
In a company, all files which are sent on the network are automatically encrypted by using AES-128 in CBC mode. **A fixed key is used, and the IV is changed once per day**. The network encryption is file-based, so that the IV is used at the beginning of every file.You managed to spy out the fixed AES-128 key, but do not know the recent IV.Today, you were able to eavesdrop two different files, **one with unidentified content (A)** and **one which is known to be an automatically generated temporary file and only contains the value 0xFF (B)**. Briefly describe how it is possible to obtain the unknown initialization vector and how you are able to determine the content of the unknown file.

Assuming that the **key is known**:
When getting the file (B) I know both the plaintext and the ciphertext. The file is made only of one block, since it is clearly of a size < 16 bytes. Knowing that the used cipher is an AES128 in CBC mode, we may obtain the IV computing the operation *dec(ciphertext) XOR plaintext**,*** since *plaintext = dec_k(ciphertext) XOR IV.*
Once obtained the IV, I can easily decrypt the file (A).

EXERCISE 5.5 Understanding Cryptography
Describe how the OFB mode can be attacked if the IV is not different for each execution of the encryption operation.

If the same IV is used for the OFB encryption, the **confidentiality may be compromised**.
If a plaintext block xj of such a message m is known, the output can be computed easily from the ciphertext block yj of the message m. This information then allows the computation of the plaintext block xj' of any other message m′ that is encrypted using the same IV.
When using the same IV, the resulting encryption with the same key is always the same!

EXERCISE 5.10 Understanding Cryptography
Sometimes **error propagation** is an issue when choosing a mode of operation in practice. In order to analyze the propagation of errors, let us assume a bit error (i.e., a substitution of a "0" bit by a "1" bit or vice versa) in a ciphertext block yi.

1. Assume an error occurs during the transmission in one block of ciphertext, let's say yi. Which cleartext blocks are affected on Bob's side when using the **ECB mode**?
2. Again, assume block yi contains an error introduced during transmission. Which cleartext blocks are affected on Bob's side when using the **CBC mode**?
3. Suppose there is an error in the **cleartext xi on Alice's side**. Which cleartext blocks are affected on Bob's side when using the **CBC mode**?
4. Prepare an overview of the effect of bit errors in a ciphertext block for the modes ECB, CBC, CFB, OFB and CTR. Differentiate between random bit errors and specific bit errors when decrypting yi.

1. In **ECB mode** only the block with the flipped bit is affected by error, since the blocks are processed separately both for encryption and decryption.

2. If a ciphertext block y_i in **CBC** is affected by error, during decryption only the subsequent cleartext block x_(i+1) will be affected, since the ciphertext block y_i will act as IV to obtain the cleartext block x_(i+1) : x_(i+1) = y_i XOR dec_k(y_(i+1)) .
   Block x_i is entirely corrupted, block x_(i+1) is corrupted in a specific pattern.
3. When the error is on the ciphertext block x_i on Alice's side, during encryption **all the blocks from y_i to the last block** will be affected, because the error is propagated since the plaintext (with the IV) concurs to the formation of the IV for the next block, and so on.
   When Bob decrypts the message, only the block x_i is affected.
4. .

Mode	Effect of error in y_i (RBE: random bit errors, SBE: specific bit errors)
ECB	RBE in decryption of y_i
CBC	RBE in decryption of y_i , SBE in decryption of y_{i+1}
CFB	SBE in decryption of y_i , RBE in decryption of y_{i+1}
OFB	SBE in decryption of y_i
CTR	SBE in decryption of y_i

EXERCISE Evaluate the truth of the following assertions:

- ECB is insecure for encrypting one single block of plaintext → ECB on a single block is equivalent to using just the block cipher over the block, hence the security mainly depends on this component. However, the encrypted block will not be randomized, hence an attacker is able to recognize when you send the message twice.
- ECB is parallelizable →**TRUE**, both for encryption and decryption
- CBC-encryption is parallelizable → **FALSE**, since the encryption of a plaintext block x_(i+1) into y_(i+1) depends both from the plaintext block x_(i+1) and from the IV that, when i > 1 is the previous ciphertext block y_i.
- CBC-decryption is parallelizable →**TRUE**, since the decryption of a ciphertext block y_(i+1) into x_(i+1) depends both from the ciphertext block y_(i+1) and from the IV that, when i > 1 is the previous **ciphertext block** y_i, and we already have it.
- In CBC decryption: a bit flip in the ciphertext corrupts only the current block →**FALSE**, it also corrupts the subsequent block, since to obtain the plaintext block x_(i+1) we need both the ciphertext block y_(i+1) and the IV that, when i > 1, is the previous ciphertext block y_i, that is corrupted. So the plaintext block x_(i+1) will be corrupted, too.
- CBC-MAC is insecure for variable-length plaintext messages→**TRUE** (see MAC with CBC mode), if attacker knows correct message-tag pairs (x, t) and (x', t'), where x has L blocks and x' has L' blocks, then he can generate a third (longer) message x'' whose tag will also be t':

$$x'' = x_1 \,||\ldots||x_L\,||\,(x'_1 \oplus t)\,||x'_2\,||\ldots||x'_{L'}$$

XOR first block of x' with t and then concatenate x with this modified x'. The resulting (x'', t') is a valid pair. This works because the xor operation with t "cancel out" the contribution from x. See slides for the proof.

EXERCISE Aes and CBC mode.

1. Describe AES128-CBC.
2. If the key is compromised, but the IV has been kept secret, is AES128-CBC still secure?
3. How would you change the answer to Q2 if the encryption was AES256-CBC?

1. AES is a block cipher that can encrypt a block of a fixed size per time. In AES128, blocks are made of 16 bytes. Aes runs in several rounds (11 rounds with 128 bit key length) and each round performs four main steps: SubBytes, ShiftRows, MixColumn and AddRoundKey. Several of these steps perform computations over Galois Fields.
   Using AES in CBC mode means that we need an IV, that can be fixed or given into the AES algorithm with the plaintext and the key. Then, during **encryption** the plaintext is XORed with the IV and the result is given to the encryption algorithm to be transformed into the ciphertext. This resulting ciphertext will act as IV in the encryption of the next block of plaintext. For **decryption**, the ciphertext is given in input with the IV. To decrypt a block of ciphertext y_(i+1), that block is given to the decryption algorithm and the output is XORed with the previous block of ciphertext y_i, or with the IV in case i = 1, to obtain the plaintext block x_(i+1).
2. **No**, if the key is compromised AES is no more secure: if an attacker know the key and the ciphertext, he can easily decrypt all the ciphertext blocks y_(i+1) by simply passing the block to the decryption algorithm and then XORing it with the previous ciphertext block y_i. Only the first block will stay secret since the IV is not known.
3. Basically nothing changes: the attack process is the same, even if the key is bigger. Keep in mind that the block size in AES is **always 128 bits**, regardless of the key length. This means that the IV in CBC will be 128 bits, regardless of the key length.

EXERCISE Iterated Symmetric Encryption

1. Describe what **iterated encryption** (aka iterated cipher) is and describe a general model. Discuss the security improvements that it is possible to obtain, and under what circumstances.
2. Describe the **Meet-in-the-Middle** attack and show how it makes double encryption as secure as single encryption. Can it be used in the case of multiple levels of encryption?

1. The main idea is to apply encryption or decryption sequences using different keys. Two popular approaches are **EEE mode:** $y = E_{k1}(E_{k2}(E_{k3}(x)))$ like in 3-DES and **EDE mode:** $y = E_{k1}(D_{k2}(E_{k1}(x)))$, where encryptions are performed with key k1 and the decryption is performed with key k2.
   If a block cipher is broken, then iterated encryption does not help significantly. On the other hand, **if the block cipher is weak due to a limited key length (like DES)** then this approach may be a way of making it stronger. Using the same key for subsequent steps (operation i and i+1) does not increase the robustness.

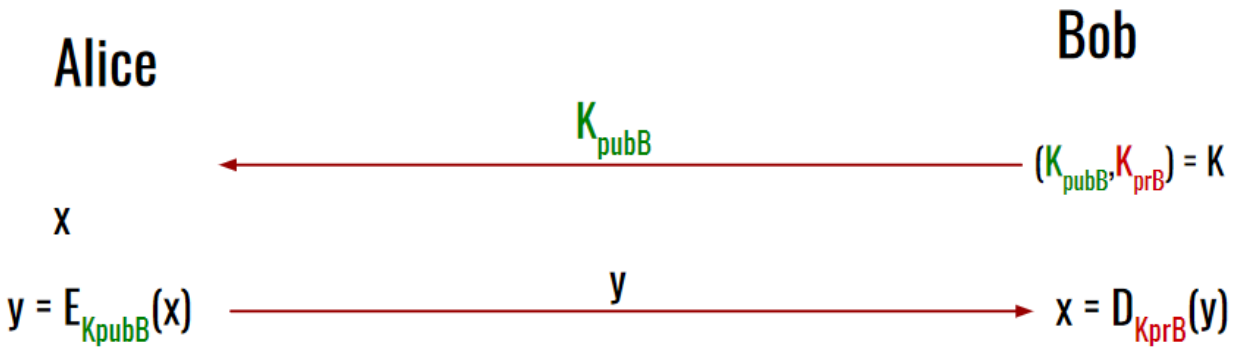
However, the “increase” of security must be considered carefully: given P keys of size N, the approach will NOT have a “key strength” equivalent to P*N. For instance, 2-DES has a key strength of just 2^57 due to MITM, even when using two keys of size 56 bits, while 3-DES has a key strength of 2^112.

2. Assuming to have a pair plaintext - ciphertext (x, y), a **Meet In The Middle** attack is:
   - 2^56 attempts: $A = \{\forall\, k_1 : E_{k1}(x)\}$
   - 2^56 attempts: $B = \{\forall k_2 : D_{k2}(y)\}$
   - Find a matching (a, b) such that $a = b$ where $a \in A, b \in B$
   - The key k1 is the one that generated a, the key k2 is the one that generated b
   - The total number of attempts is 2^56 + 2^56 = 2^57.

Asymmetric Cryptography

Asymmetric cryptography idea is like a mailbox: everybody can put letters in, but only the owner can read them.

The **key k** is splitted into a **public key** k_{pub} used for encryption and a **private key** k_{pr} used for decryption.
Bob, the owner of the key, publishes his public key k_{pub} , so **anyone can encrypt** a message using k_{pub} .
Then the message is sent back to Bob, that is the **only one that can decrypt** a message encrypted with k_{pub} using the decryption private key k_{pr}**:** these keys have been generated in pair, so only the owner of the private key can decrypt a message that has been encrypted with the public key.



With asymmetric cryptography the **problem of distributing the key** is solved. However this is not entirely true as we **need to authenticate** the public key.

Public-Key Cryptography can be used for:

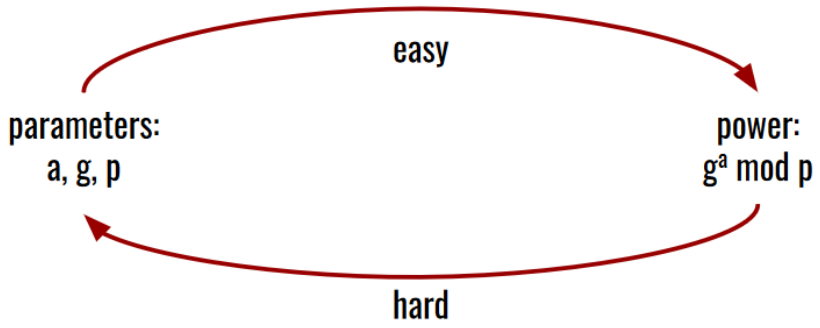
- **Key distribution** (using Diffie-Hellman key exchange or RSA) without a pre-shared secret key.
- **Nonrepudiation** and **Digital Signatures** (RSA, DSA, ECDSA) to provide message integrity.
- **Identification**, using challenge-response protocols with digital signatures.
- **Encryption** (RSA, Elgamal), however the computational effort is very high - about 1000 times slower than symmetric algorithms!

Diffie-Hellman Key Exchange

A simple public-key algorithm is Diffie-Hellman **key exchange**. This protocol enables two users to establish a secret key using a public-key scheme **based on discrete logarithms**. The protocol is secure only if the authenticity of the two participants can be established**.**

The purpose of the algorithm is to enable two users to securely exchange a key that can then be used for subsequent encryption messages. The algorithm itself is limited to the exchange of secret values.

The Diffie-Hellman algorithm depends for its effectiveness on the **difficulty of computing discrete logarithms**: given some parameters a, i, p it is **easy** to compute $b = a^i (mod\ p)$but, given b, it is **hard** to find a, i, p since there is **no trapdoor** in DH.
A **trapdoor function** is a function that is easy to compute in one direction, yet difficult to compute in the opposite direction (its inverse) without special information, called “trapdoor”. DH does not have this trapdoor.



DISCRETE LOGARITHM - Let a *primitive root* of a prime number p as one whose powers modulo p generate all integers from 1 to $p - 1$. That is, if a is a *primitive root* of the number p, then the numbers $a\ mod\ p, a^2\ mod\ p, \ldots, a^{p-1} mod\ p$ are distinct and consist of the integers from 1 through $p - 1$ in some permutation.
For any integer b and a primitive root a of prime number p, we can find an unique exponent i such that:

$$b = a^i\ (mod\ p) \text{ where } 0 \leq i \leq (p - 1)$$

The exponent i is referred to as the **discrete logarithm** of b for the base a, mod p. We express this value as $dlog_{a,p}(b)$.

Global Public Elements	
q	prime number
α	$\alpha < q$ and α a primitive root of q

User A Key Generation	
Select private X_A	$X_A < q$
Calculate public Y_A	$Y_A = \alpha^{X_A} \bmod q$

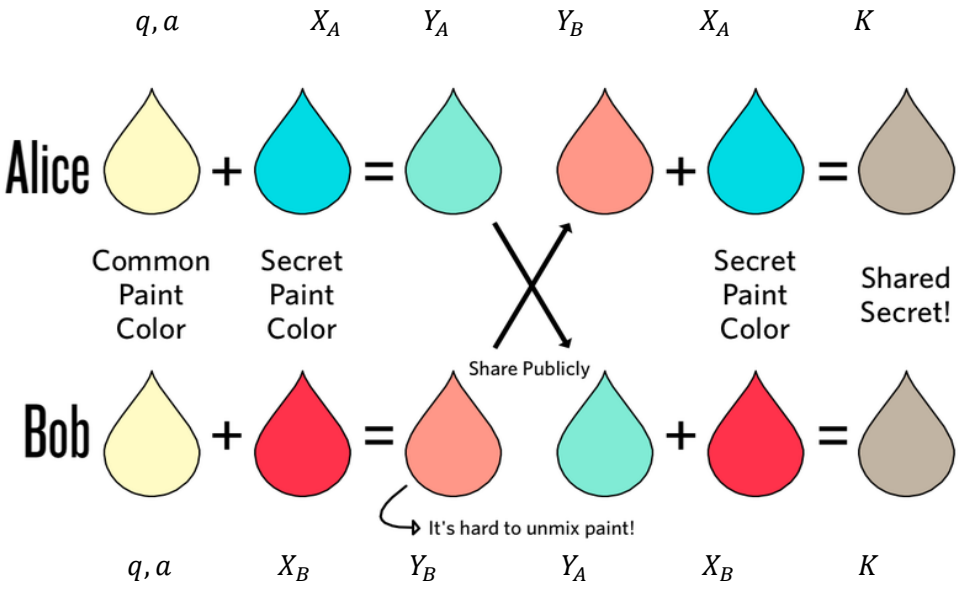
User B Key Generation	
Select private X_B	$X_B < q$
Calculate public Y_B	$Y_B = \alpha^{X_B} \bmod q$

Calculation of Secret Key by User A

$K = (Y_B)^{X_A} \bmod q$

Calculation of Secret Key by User B

$K = (Y_A)^{X_B} \bmod q$



- q and a are **public**
- X_A and X_B are the **private keys**
- $Y_A = a^{XA} mod\ q$ and $Y_B = a^{XB} mod\ q$ are the **public keys**
- $K = a^{XA * XB}\ mod\ q$ is the **shared secret** between A and B

Remind that DH is **not for encryption** but for key exchange: the purpose is once both A and B know the key k, they can perform encryption with a symmetric scheme (e.g. AES).

Security of Diffie-Hellman Key Exchange

SMALL SUBGROUP CONFINEMENT ATTACK - If we just consider any generator of $Z^*{}_q$ then an attacker can look at the subgroups generated by $Y_A = a^{XA} mod\ q$ and $Y_B = a^{XB} mod\ q$ and so restrict the set of "possible" keys when one of these two subgroups is small.

Example - $Z^*{}_{19}$, $a = 2,\ X_A = 6, X_B = 11$
We have that $\forall r:\ (a^{XA} mod\ q)^r = (2^6\ mod\ 19)^r = 7^r = \{1, 7, 11\}$
Hence, we can easily get that the key $K = a^{XA * XB}\ mod\ q$ can be only one of $\{1, 7, 11\}$.
In practice, we care also for properties on the subgroups of $Z^*{}_q$.

Analyzing some properties of **cyclic groups**, it is possible to deduce that if q is prime, then $|G| = q - 1$ which is an even number. Hence, we have subgroups of size:

- 1
- 2
- $(q-1)/2$
- $q - 1$
- … any other divisor of $|G|$

How to **avoid small subgroups**? We can require that the prime q defining the group **safe-prime**, i.e. $q = 2p + 1$, where p is a prime. Under this assumption, the divisors of $|G| = q - 1 = 2p + 1 - 1 = 2p$ are:

- 1
- 2
- p
- $2p$

So we should define rules to avoid groups of size 1 and 2 and require that p is very large, i.e. more than 1024 bits.
q must be large in order to have p large and to prevent attacks such as Pohling-Hellman (based on CRT), Pollard's Rho, Index Calculus (probabilistic) and others.

PERFECT FORWARD SECRECY - If the secrets X_A, X_B, K are compromised, then other communication sessions will not be compromised. The value of forward secrecy is that it

protects past communication. This reduces the motivation for attackers to compromise keys. For instance, if an attacker learns a long-term key, but the compromise is detected and the long-term key is revoked and updated, relatively little information is leaked in a forward secure system.

PASSIVE ATTACKER - A passive attacker knows $a, q, Y_A = a^{XA} \bmod q, Y_B = a^{XB} \bmod q$.
He can easily compute $a^{XA+XB} \bmod q$ but not $K = a^{XA*XB} \bmod q$.
Why? To build shared secret it needs to recover X_A (or X_B). One way of breaking DH is to solve **DLP** which is believed to be hard to break. However, there could be other ways of breaking DH that do not necessarily need to solve DLP.

ACTIVE ATTACKER - An active attacker can perform a **Man-In-The-Middle (MITM)** attack:

1. Adversary fakes Bob identify and performs DH with Alice
2. Adversary fakes Alice identify and performs DH with Bob
3. A and B think that they have a common shared key, instead they are sharing a secret key with the Adversary.

Is it possible to be safe against a MITM attack when using Diffie-Hellman? Not too much can be done. What can be done is to make **authentication** between the two parties. With a strong authentication mechanism, DH will be a good solution for the key exchange. Without authentication, DH is vulnerable to the MITM attack.

Diffie-Hellman Key Exchange with three parties

Alice, Bob and Carol could participate in a DH agreement as follows, with **all operations taken to be modulo** q:

1. A, B and C agree on the parameters q and a.
2. The parties generate their private keys X_A, X_B, X_C.
3. A computes a^{XA} and sends it to B.
4. B computes a^{XA*XB} and sends it to C.
5. C computes $a^{XA*XB*XC} = K$ and uses it as her secret.
6. B computes a^{XB} and sends it to C.
7. C computes a^{XB*XC} and sends it to A.
8. A computes $a^{XB*XC*XA} = a^{XA*XB*XA} = K$ and uses it as her secret.
9. C computes a^{XC} and sends it to A.
10. A computes $a^{XC*XA} = a^{XA*XC}$ and sends it to B.
11. B computes $a^{XC*XA*XB} = a^{XA*XB*XC} = K$ and uses it as his secret.

To easily remember, notice that there is a **circular exchange**: $(A \rightarrow B \rightarrow C), (B \rightarrow C \rightarrow A), (C \rightarrow A \rightarrow B)$.

This protocol is **not secure against a MITM attack**. The main issue is that, similar to the traditional DHKE, **there is no authentication of the public keys sent by A, B and C**. Hence

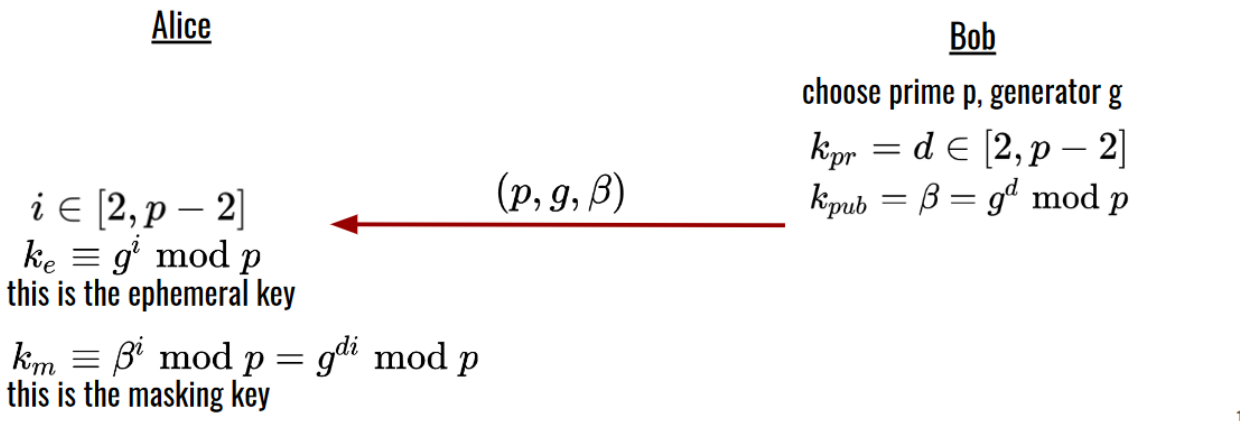
an attacker can intercept messages from one or more parties, replacing them with new messages and the other involved parties has no way of detecting that the attacker is involved in the key exchange.

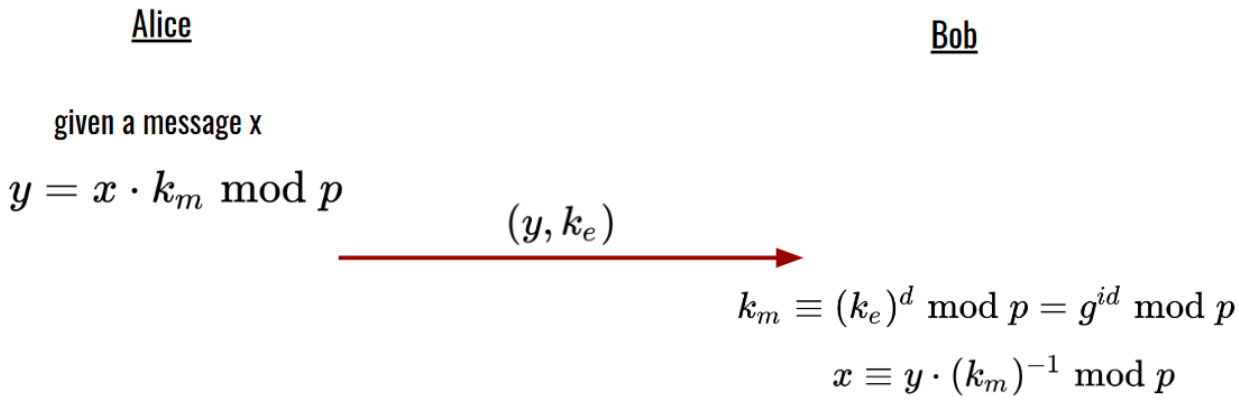
If authentication is performed at least for some entities (e.g. C is a server), then the attacker cannot impersonate these entities but only the remaining ones. This could be a trade-off that some applications may consider.

Elgamal Encryption Scheme

It uses the same idea of DH but instead of doing key exchange, this does encryption.

An extension of the DHKE protocol that allows also encryption. Only a few details have been changed, optimizing the order and number of messages exchanged between Alice and Bob.





RSA Algorithm (Rivest, Shamir, Adleman)

RSA is a **block cipher** in which the plaintext and the ciphertext are integers between 0 and $n - 1$ for some n.
The block size must be less than or equal to $log_2(n) + 1$. In practice, the block size is i bits, where $2^i < n \leq 2^{i+1}$.
Encryption and decryption are of the following form, for some plaintext block M and ciphertext block C:

$$C = M^e \bmod n$$
$$M = C^d \bmod n = (M^e)^d \bmod n = M^{ed} \bmod n$$

- Both sender and receiver must know the value of n.
- The sender knows the value of e, and only the receiver knows the value of d.
- This is a public-key encryption algorithm with **public key** $PU = \{e, n\}$ and **private key** $PR = \{d, n\}$

For a correct behavior of the algorithm it is needed to find a relationship of the form

$$M^{ed} \bmod n = M$$

and

$$ed \bmod \phi(n) = 1$$

This assumption holds if e and d are **multiplicative inverse** modulo $\phi(n)$, and this is true if and only if d (and so does e) is **coprime** to $\phi(n)$, i.e. $gcd(d, \phi(n)) = 1$.

Thus, the **range of the exponent** is $(1, \phi(n))$. So, for example, if $p = 13, q = 17$ we obtain that $\phi(n) = (13 - 1) \times (17 - 1) = 192$. So the range for the exponent is $1 < e < 192$.

Key Generation Alice

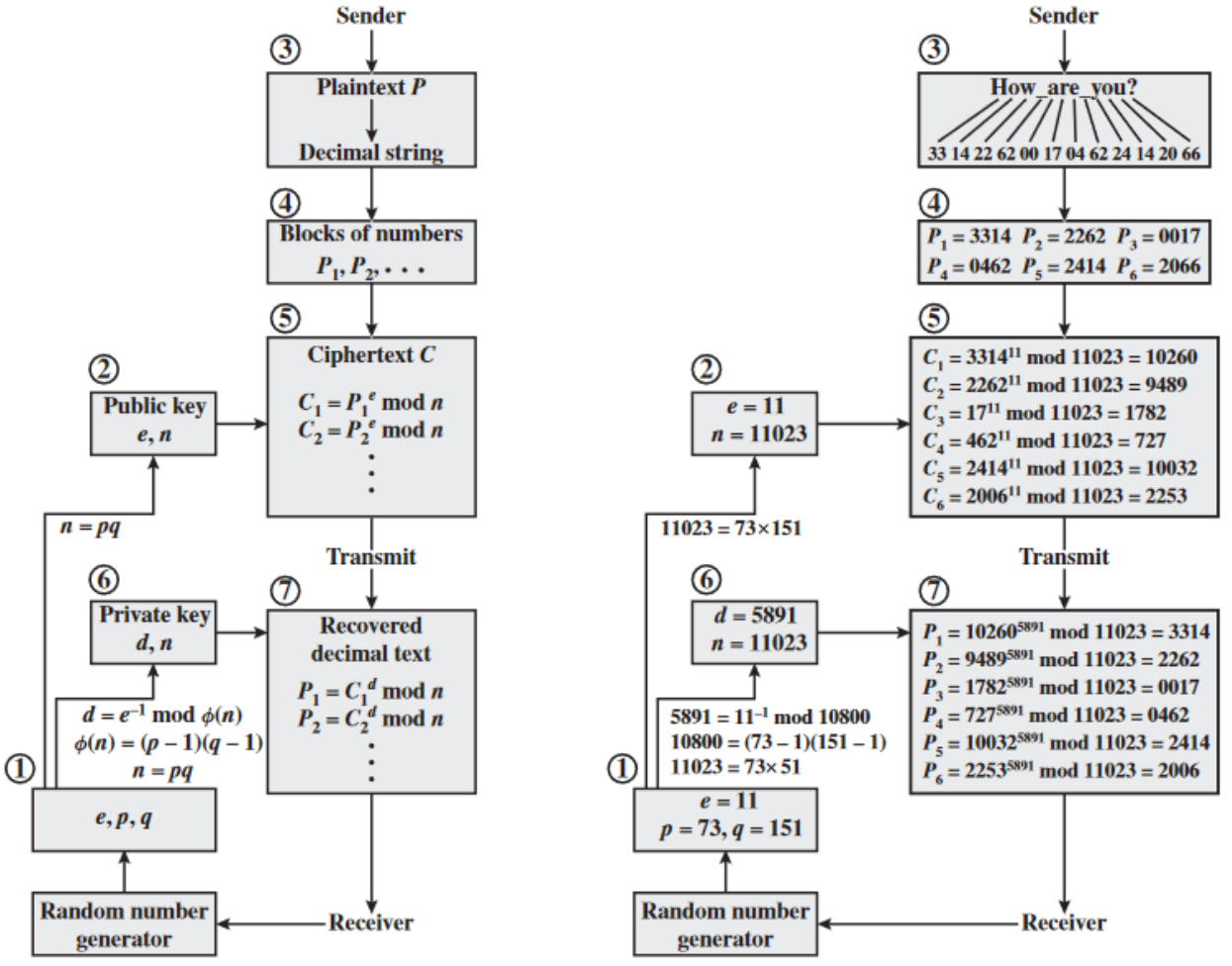
Select p, q	p and q both prime, $p \neq q$
Calculate $n = p \times q$	
Calcuate $\phi(n) = (p-1)(q-1)$	
Select integer e	gcd $(\phi(n), e) = 1; 1 < e < \phi(n)$
Calculate d	$d \equiv e^{-1} \pmod{\phi(n)}$
Public key	$PU = \{e, n\}$
Private key	$PR = \{d, n\}$

Encryption by Bob with Alice's Public Key

Plaintext:	$M < n$
Ciphertext:	$C = M^e \bmod n$

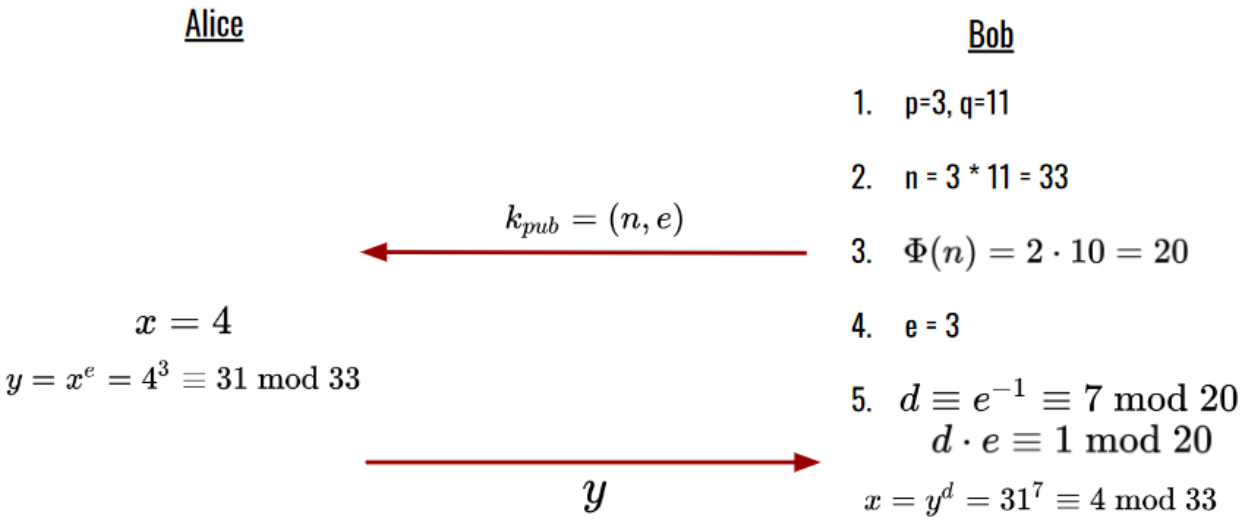
Decryption by Alice with Alice's Public Key

Ciphertext:	C
Plaintext:	$M = C^d \bmod n$



(a) General approach (b) Example

Simple example, where x = plaintext message, y= encrypted message.



What if the message block M is larger than n?
If the message $M$is bigger than n, i.e. $|M| > n$, after decryption we don't get the original message M, but:

$$M' = d_{k_{pr}}(C) = C^d mod\, n$$

Which by definition is smaller than n. Hence $M' != M$.

Some Properties of RSA

We need to perform exponentiation in both encryption and decryption: there is a design choice since **we can make encryption faster (choosing a small value for e**) **but then we get a large d (hence decryption is expensive**), or we can choose to make d small but e large (in this case we have to perform key generation a little bit different), getting slow encryption but fast decryption. However, having a small d is completely unsafe.
A common choice is $e = 65537$, that is equal to $2^{16} + 1$, whose binary representation is 10000000000000001, that is a good prime number due to its short Hamming Weight.
So it is important to use an **efficient algorithm for exponentiation**.
To cope with this problem it is possible to use a **Square-And-Multiply** algorithm: scan all the bits of the exponent from left to right. In every iteration we square the base. If the current bit is one, then we multiply
Here are **two algorithms** for Square-And-Multiply (With different terminology. Pick your best!)

```
static long fastExp(int base, int exp) {
    long f = 1;
    long b = base;
    while(exp > 0) {
        int lsb = 0x1 & exp;
        exp >>= 1;
        if(lsb) f *= b;
        b *= b;
    }
    return f;
}
```

```
c ← 0; f ← 1
for i ← k downto 0
    do  c ← 2 × c
        f ← (f × f) mod n
    if  b_i = 1
        then c ← c + 1
             f ← (f × a) mod n
return f
```

Note: The integer b is expressed as a binary number $b_k b_{k-1} \dots b_0$.
a is the base, b is the exp.

Proof of Correctness based on Fermat's Little Theorem

[LINK]

$$x \equiv x^{ed} \bmod (p \cdot q)$$

Based on the Chinese Remainder Theorem this is true when:

$$x \equiv x^{ed} \bmod p$$
$$x \equiv x^{ed} \bmod q$$

Since:

$$ed \equiv 1 \bmod (p-1)(q-1)$$
$$ed \equiv 1 + k \cdot (p-1)(q-1)$$

Therefore:

$$\begin{aligned} x^{ed} &\equiv x \cdot x^{k(p-1)(q-1)} \bmod p \\ &\equiv x \cdot (x^{p-1})^{k(q-1)} \bmod p \\ &\equiv x \cdot 1^{k(q-1)} \bmod p \\ &\equiv x \bmod p \end{aligned}$$

(Fermat's Little Theorem)

Symmetric proof for showing that $x \equiv x^{ed} \bmod q$

Security in RSA

Is RSA secure? The main assumption behind RSA is that it should be hard to compute d or $\phi(n)$ or p or q. So it would be good to select p and q as **large distinct prime numbers** (more than 1024 bits), to chose $e \geq 3$and that $(p-1)$ and $(q-1)$ should have large prime factors otherwise Pollard's rho algorithm is able to perform factorization efficiently.
Still this 'textbook' implementation of RSA is not secure and suffers from different problems. (e.g. No randomization is provided for the ciphertext C)

Four possible approaches to attacking the RSA algorithm are

- **Brute force**: this involves trying all possible private keys.
- **Mathematical attacks** : there are several approaches, all equivalent in effort to factoring the product of two primes.
- **Timing attacks**: these depend on the running time of the decryption algorithm.
- **Chosen ciphertext attacks**: this type of attack exploits properties of the RSA algorithm.

Math Attack - Factoring Problem - Factoring $n = p \times q$ should be hard, unless p and q have some bad properties.
It would be good to choose p and q **large enough** (100 digits each).
Make sure p and $q$are **not close** together.
Make sure both $(p-1)$ and $(q-1)$ should have **large prime factors** to foil Pollard's rho algorithm.
Example: given n, e, p, q it is easy to compute d. Also given n, e if you factor n then you can compute $\phi(n)$ and so d.

Avoid specific 'corner case' messages - Some messages such as $M = 0, 1, (n-1)$ are easy to decode, since RSA(M) = M.
If both M and e are small, then we may have that $M^e < n$, and the attacker can easily get the plaintext by computing e^{th}root of the encrypted message. A **solution** could be to add non zero bytes to avoid small messages.

Chinese Remainder Theorem Attack - If k ciphertexts of the **same plaintext** $M$are encrypted with the **same exponent** e using different (p, q), then the message can be decrypted using the Chinese Remainder Theorem (CRT). A **solution** could be to add random bytes to avoid equal messages.

Small message space - Given a ciphertext C, if the message space is small (e.g. k values), then the adversary can encrypt all k values and then compare results with the ciphertext C. A **solution** could be to add random strings in the message.

Same n for different users - If more users have the same n (but different e and d), one user could compute p, q starting from his e, d, n and then the user could easily discover the secret key of another user, given his public key. **Solution:** each person chooses a different n.

Chosen Ciphertext Attack - Adversary wants to decrypt $C = M^e \ mod \ n$, so:

1. Adv. computes $X = (C * 2^e) \ mod \ n$
2. Adv. uses X as a chosen ciphertext and asks the the decryption for $Y = X^d \ mod \ n = (C * 2^e)^d mod \ n$.

But since $X = (C * 2^e) \ mod \ n = (C \ mod \ n) * (2^e \ mod \ n) = (M^e \ mod \ n) * (2^e \ mod \ n) = (2M)^e \ mod \ n$
Then the Adversary got $Y = 2M$, that can be divided by 2 e.g. with the use of the multiplicative inverse in modular arithmetic. **Solution**: verify the structure of the message to decrypt before.

RSA: How to mitigate/solve most of these attacks?

Textbook RSA is unsafe. Textbook Elgamal is unsafe. In practice, we should always use a standard that provides rules on how to use a PK scheme for real-world usages:

Solution: PKCS (Public Key Cryptography Standards)

PKCS#1: RSA Cryptography Standard
PKCS#3: Diffie Hellman Key Agreement Standard
PKCS#11: Elliptic Curve Cryptography Standard

PKCS#1 define an encryption scheme called **RSAES-OAEP**, that improves RSA with Optimal Asymmetric Encryption Padding (OAEP).
The idea of OAEP is to preprocess the message getting a new one and then encrypt the new message. This should be completely invertible and easy to invert.

RSA AND DATA INTEGRITY: OAEP

- Padding scheme often used together with RSA encryption
  - introduced by Bellare and Rogaway: "Optimal Asymmetric Encryption - How to Encrypt with RSA" 1994, 1995 (http://cseweb.ucsd.edu/users/mihir/papers/oae.pdf)
- OAEP uses a pair of random oracles G and H to process the plaintext prior to asymmetric encryption
  - a random oracle is a mathematical function mapping every possible query to a random response from its output domain.
  - when combined with RSA it is secure under chosen plaintext/ciphertext attacks
- OAEP satisfies the following two goals
  1. Add an element of randomness which can be used to convert a deterministic encryption scheme (e.g., traditional RSA) into a probabilistic scheme.
  2. Prevent partial decryption of ciphertexts (or other information leakage) by ensuring that an adversary cannot recover any portion of the plaintext without being able to invert the trapdoor one-way permutation f.

Optimal Asymmetric Encryption Padding (OAEP)

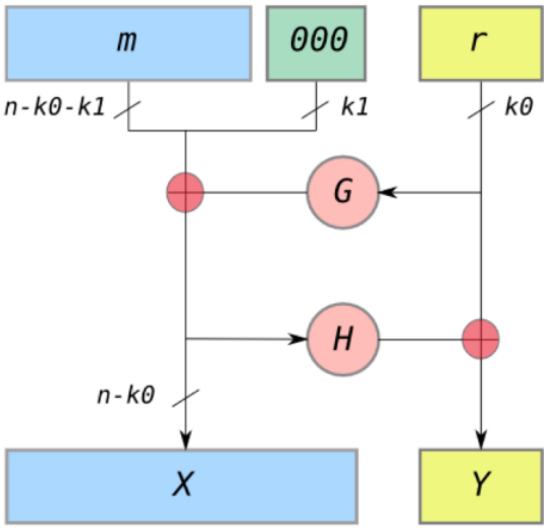
n = number of bits in RSA modulus
k0 and k1 = integers fixed by the protocol
m = plaintext message, a (n − k0 − k1)-bit string
G and H = cryptographic hash functions fixed by the protocol

To encode

- messages are padded with k1 zeros to be n − k0 bits in length.
- r is a random k0-bit string
- G expands the k0 bits of r to n − k0 bits.
- X = m00..0 ⊕ G(r)
- H reduces the n − k0 bits of X to k0 bits.
- Y = r ⊕ H(X)
- output is X‖Y

To decode

- recover the random string as r = Y ⊕ H(X)
- recover the message as m00..0 = X ⊕ G(r)



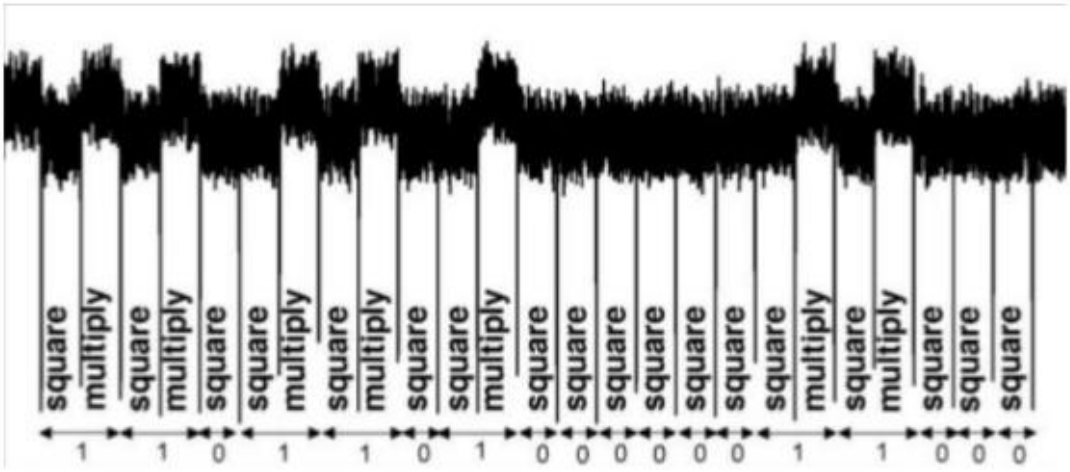
*G and H are **identical** in PKCS#1 and **MGF1** (mask generation function, a hash with programmable output-size)*

46

OAEP "All or Nothing Security" - To recover m you must recover the entire X and the entire Y. X is required to recover r from Y, and r is required to recover m from X.
Since any bit of the cryptographic hash completely changes the result, the entire X and the entire Y must both be completely recovered.

Another feasible attack on RSA is based on power analysis, by analyzing the CPU usage that increases when a multiplication is done.

Power analysis can recover the key when performing exponentiation with the private key:



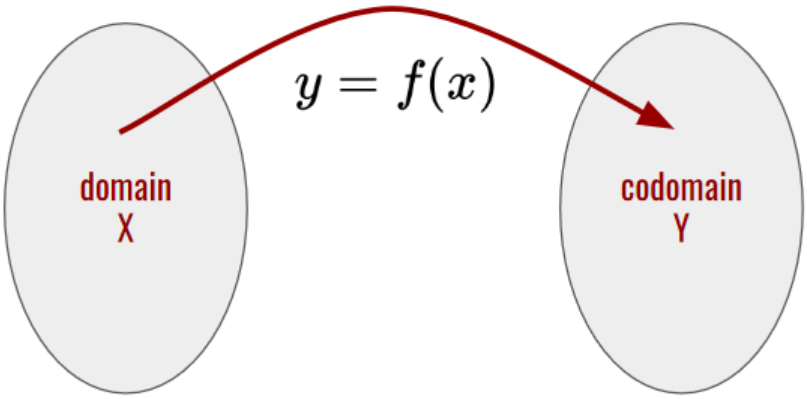
How to prevent this attack?
Modify square-and-multiply in order to take constant time (power) at each iteration, regardless of the exponent bit value: e.g., add a dummy multiplication in s-a-m. Still, real-world implementations (e.g., based on Montgomery Reduction) are weak to more advanced power analyses.
Another approach is **message randomization** (bliding) where, before decryption, the system picks a random r and computes r^e, then decrypts $C * r^e$ obtaining $M * r$, and from then divides it by r to get back M.

How to prevent effective power analysis or other timing attacks during exponentiation?
Always **use up-to-date implementations** of RSA (e.g., openSSL, libreSSL) that are specifically designed to prevent these attacks. They are frequently updated to prevent the most cutting-edge attacks.

RsaCtfTool can test many well-known attacks for RSA.
Yafu o **msieve** can be used for performing factorization of “small” numbers. Libraries such as **SageMath** have integrated several algorithms for factorization.

Hash Functions

Hash functions are an important cryptographic primitive and are widely used in protocols. They compute a **digest** of a message which is a **short, fixed-length bit-string**. For a particular message, the message digest, or **hash value**, can be seen as the fingerprint of a message, i.e., a **unique representation of a message**.
Unlike all other crypto algorithms, hash functions **do not have a key**. The use of hash functions in cryptography is manifold: Hash functions are an essential part of digital signature schemes and message authentication codes.
Hash functions are also widely used for other cryptographic applications, e.g., for storing password hashes or key derivation.



y is often called:
- hash value
- hash code
- digest

Hash Functions are **non invertible functions.**
The goal is to **map** large domains into smaller codomains: $|X| >> |Y|$.
It is crucial to **minimize collisions**, i.e. different values in the domain mapped into the same value of the codomain.

Hash functions can also be used to **guarantee data integrity**, maybe using them to represent a large amount of data in a compact value. Changing even one bit in the data should make the value of the hash change!

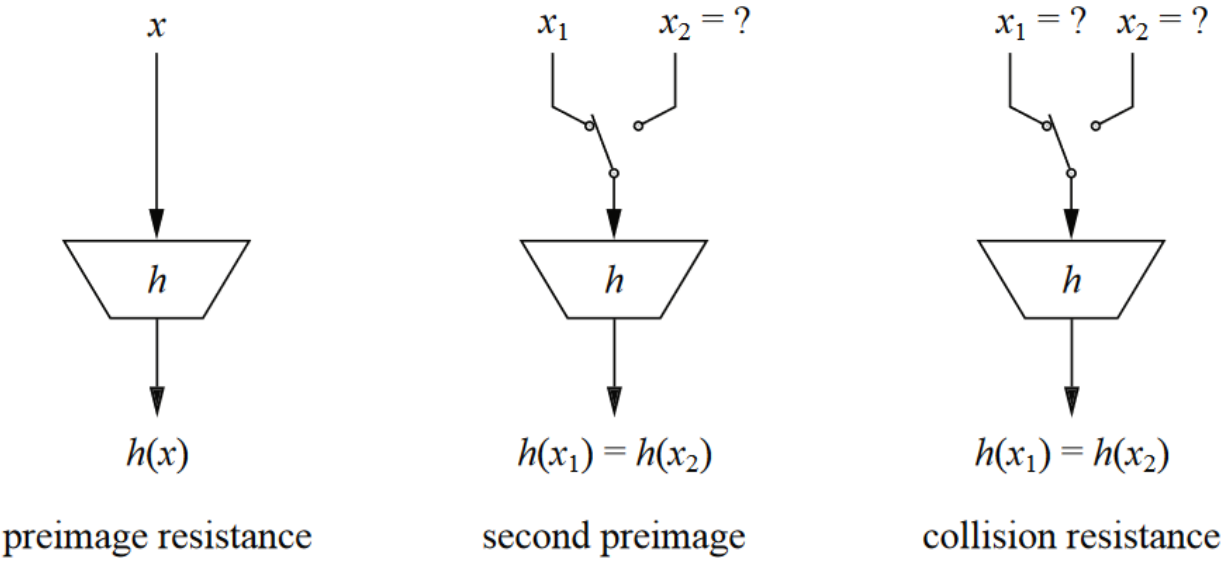
PROPERTIES OF A CRYPTOGRAPHIC HASH FUNCTION

- **Arbitrary input length** : we would like to process even large files (e.g., an entire hard disk). Notice that encryption schemes often impose restriction on the input length.
- **Fixed and short output length :** in several application contexts, we would like to have a compact output value given (even very large) input. Notice that encryption often has an output length that is equal or larger than the input length.
- **Efficiency :** fast even when processing large inputs. Notice that encryption schemes can be very slow on large files.
- **Secure** : a small change in the input has to generate a big change in the hash.
- **Non-reversible** : given $h(x)$, it is unfeasible to determine x.

Security Requirements of Hash Functions

Since hash functions do not use keys, they have to implement three central properties in order to be secure:

1. **Preimage Resistance** (or one-wayness) : for a given output $y = h(x)$, it is computationally infeasible to find any input x such that $h(x) = y$. This means that $h(x)$ is **one-way**.
2. **Second Preimage Resistance** (or weak collision resistance) : given x_1 and thus $h(x_1)$, it is computationally infeasible to find any x_2 such that $h(x_1) = h(x_2)$. This is important to avoid **substitution attacks**. The attacker can generate x_2.
3. **Collision Resistance** (or strong collision resistance) : it is computationally infeasible to find any pairs (x_1, x_2) where $x_1 \neq x_2$ such that $h(x_1) = h(x_2)$. Here the attacker can generate both x_1, x_2, i.e. two inputs with the same hashcode.



Hash Functions and collisions

Is it possible to avoid collisions? **No**, since the domain is larger than the codomain.

DIRICHELET'S DRAWER PRINCIPLE - "if you have 10 socks and only 9 drawers, then at least one drawer will contain more than one sock".

PIGEONHOLE PRINCIPLE - "if you have 20 pigeons but only 19 boxes, then at least one box will contain at least two pigeons".

Since the output of every hash function has a fixed bit length, say n bit, there are "only" 2^n possible output values. At the same time, the number of inputs to the hash functions is infinite so that multiple inputs must hash to the same output value.
In practice, each output value is equally likely for a random input, so that weak collisions exist for all output values.
Since weak collisions exist in theory, the next best thing we can do is to **ensure that they cannot be found in practice**.

BRUTE FORCE ATTACK - Given $(x, h(x))$, and adversary can always perform a brute force attack to find $(x', h(x'))$ such that $x \neq x'$ and $h(x) = h(x')$. This attack can be used for the Second Preimage (or **weak collision resistance**).
If no properties for the specific hash function $h(\,)$are found, then the adversary can try to compute $h(x')$ for **any possible** x' **until a collision is found**.
The expected complexity of the attack is 2^n, where n is the number of bits of the output.
Hence, n must be large enough to make the attack infeasible, e.g. n = 256.

Collision Resistance and the Birthday Attack

An hash function is **collision resistant** or **strong collision resistant** if it is computationally infeasible to find two different inputs $x_1 \neq x_2$ with $h(x_1) = h(x_2)$.
As seen before, collisions always exist. The question is how difficult it is to find them. An intuitive answer to this question is that it is as difficult as finding second preimages with a brute force attack.
However, if the hash function has an output of length of 80 bits, instead of checking for 2^{80} messages, it turns out that an attacker only needs about 2^{40} messages! This is explained by the **birthday paradox.**

BIRTHDAY PARADOX - In second preimage, the simplified problem would be "if we have a party, how many people should I invite to have one person's birthday on a specific date XYZ?". In this case, referring to the **Strong Collision Resistance**, on average, we expect that the answer is 365 people. Here is why, generalized over hash functions with n bit output.

$$P(\text{no collision}) = \left(1 - \frac{1}{2^n}\right)\left(1 - \frac{2}{2^n}\right)\cdots\left(1 - \frac{t-1}{2^n}\right)$$
$$= \prod_{i=1}^{t-1}\left(1 - \frac{i}{2^n}\right)$$

After some simplifications (see 11.2.3 of "Understanding Cryptography") we get that for 0.5 probability of a collision:

$$t \approx 2^{(n+1)/2}\sqrt{\ln\left(\frac{1}{1-\lambda}\right)} \qquad \lambda = 1 - Pr(\text{no collision})$$

In practice, we can see that the attack complexity is roughly the square root of the output space.
Example : output length is n = 80 bits, output space is 2^{80}, then the expected attempts to find a collision with probability 0.5 is $\sqrt[2]{2^{80}} \simeq 2^{40}$.
Hence, if want to have an expected complexity of at least 2^{80}, we need actually an output space of at least 2^{160}.

Bits	Possible outputs (H)	Desired probability of random collision (2 s.f.) (p)									
		10^{-18}	10^{-15}	10^{-12}	10^{-9}	10^{-6}	0.1%	1%	25%	50%	75%
16	2^{16} (~6.5×10^{4})	<2	<2	<2	<2	<2	11	36	190	300	430
32	2^{32} (~4.3×10^{9})	<2	<2	<2	3	93	2900	9300	50,000	77,000	110,000
64	2^{64} (~1.8×10^{19})	6	190	6100	190,000	6,100,000	1.9×10^{8}	6.1×10^{8}	3.3×10^{9}	5.1×10^{9}	7.2×10^{9}
128	2^{128} (~3.4×10^{38})	2.6×10^{10}	8.2×10^{11}	2.6×10^{13}	8.2×10^{14}	2.6×10^{16}	8.3×10^{17}	2.6×10^{18}	1.4×10^{19}	2.2×10^{19}	3.1×10^{19}
256	2^{256} (~1.2×10^{77})	4.8×10^{29}	1.5×10^{31}	4.8×10^{32}	1.5×10^{34}	4.8×10^{35}	1.5×10^{37}	4.8×10^{37}	2.6×10^{38}	4.0×10^{38}	5.7×10^{38}
384	2^{384} (~3.9×10^{115})	8.9×10^{48}	2.8×10^{50}	8.9×10^{51}	2.8×10^{53}	8.9×10^{54}	2.8×10^{56}	8.9×10^{56}	4.8×10^{57}	7.4×10^{57}	1.0×10^{58}
512	2^{512} (~1.3×10^{154})	1.6×10^{68}	5.2×10^{69}	1.6×10^{71}	5.2×10^{72}	1.6×10^{74}	5.2×10^{75}	1.6×10^{76}	8.8×10^{76}	1.4×10^{77}	1.9×10^{77}

Result for n = 64 and p = 0.5 is called **birthday bound**.

STRONG COLLISION IMPLIES WEAK COLLISION

Proof. we show that $\neg \text{weak} \Rightarrow \neg \text{strong}$

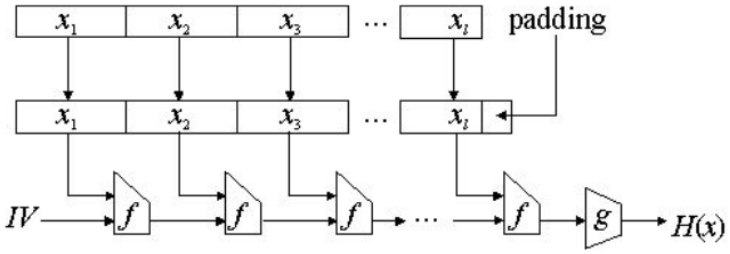
Idea. Pick randomly x, compute h(x), find x such that x != x' and h(x)=h(x'). (x, x') shows that the hash function is not strong collision resistant.

More formally:

- suppose there is polynomial algorithm A_h: $A_h(x) = x'$ such that $h(x) = h(x')$
- construct a polynomial algorithm $B_h() = (x, x')$: randomly picks x and returns $(x, A_h(x))$

Merkle - Damgard construction

Several cryptographic hash function takes as input a fixed block size (e.g., 256 bits). To fulfill the requirement on arbitrary input length, we can use the Merkle-Damgård Construction (MDC):



- IV is usually constant
- if hash function f is collision resistant, then its MDC extension is collision resistant

EXAMPLE Exam 2015 Feb 10th
Q3.2 - If the underlying hash function maps 256b blocks into 128b blocks, how many rounds are required for hashing a 140KB file?

Since the output of a compression by f must be of the **IV of the next** f, and the **IV and the input block must be of the same size**, in each compression of the chain we have to give 128b from the IV and 128b from a chunk of the file.

140 KB = 140 x 1000 x 8 = 112000 bit is the size of the file in bits.

We have to take blocks of size 128 bits, since the other (256 - 128 =) 128 bits are taken from the IV. Thus, we will need 112000 / 128 = **875 rounds** for hashing a 140KB file.

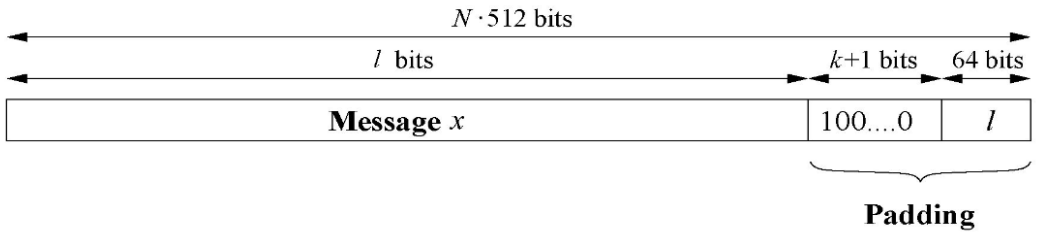
Real World Cryptographic Hash Functions

- MD (Message Digest) family:
  - MD4 by Ronald Rivest (1990): first collision attack in 1995. In 2007, an attack can generate collisions in less than 2 MD4 hash operations. A theoretical preimage attack also exists.
  - MD5 by Ronald Rivest (1992): collision attack in seconds on a 2.6 GHz Pentium 4 processor (complexity of 2^24.1). A chosen-prefix collision attack that can produce a collision for two inputs with specified prefixes within seconds, using off-the-shelf computing hardware (complexity 2^39). Pre-image attack still hard (2009: $2^{123.4}$)
- SHA (Secure Hash Algorithm) family:
  - SHA-1 by NSA (1995): the most popular cryptographic hash function, inspired by MD algorithms. Since 2005 considered not fully secure, Google was able to generate a collision in 2017. As of 2020, chosen-prefix attacks against SHA-1 are now practical.
  - SHA-3: current solution part of a standard, proposed in 2015 in response to an open call from NIST (similar to AES).

SHA-1

An interesting interpretation of the SHA-1 algorithm is that the compression function **works like a block cipher**, where the input is the previous hash value H_{i-1} and the key is formed by the message block x_i. The actual rounds of SHA-1 are in fact quite similar to a Feistel block cipher.
SHA-1 produces a 160-bit output of a message with maximum length of 2^{64} bit.
Before the computation, the algorithm has to preprocess the message.
In the actual computation, the compression function processes the message in 512-bit chunks.
The compression function consists of 80 rounds which are divided into four stages of 20 rounds each.

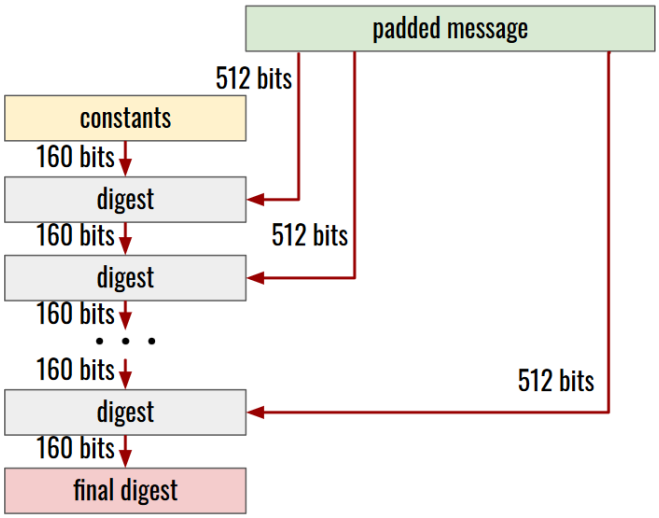
- input length: up to 2^{64}
- block size: 512 bits
- output length: 160 bits
- original message is padded with (10...00) and input length (l) to reach a multiple of block size:



- the padded message is then split in blocks of 512 bits

It is based on Merkle-Dåmgard Construction:

- each digest stage is composed by 80 rounds
- initial digest is constant and composed by five 32 bit constants {A, B, C, D, E}, which are called registers and are updated by each digest computation



Each message block $x_i$is processed in four stages with 20 rounds each as shown in the figure below.

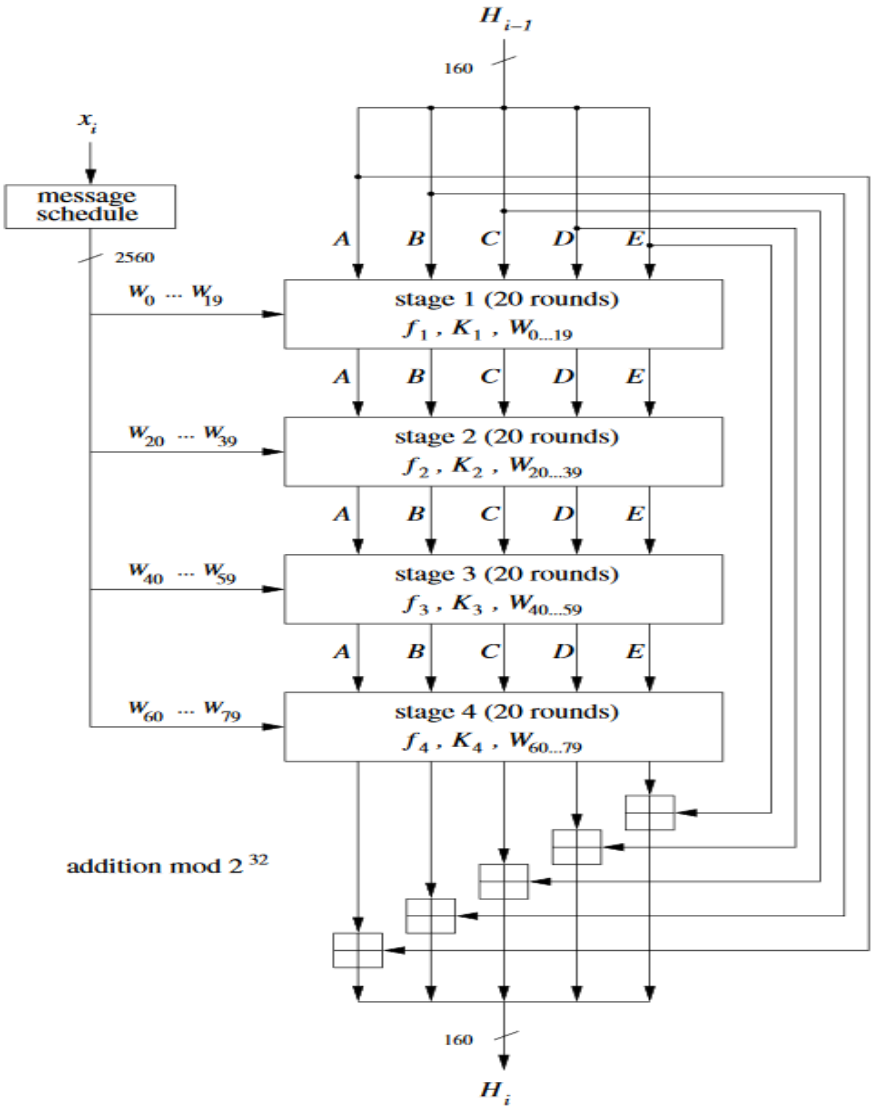
The algorithm uses:

- A message schedule which computes a 32-bit word $W_0, W_1, \ldots, W_{79}$ for each of the 80 rounds. The words W_j are derived from the 512-bit message block as follows:

$$W_j = \begin{cases} x_i^{(j)} & 0 \le j \le 15 \\ (W_{j-16} \oplus W_{j-14} \oplus W_{j-8} \oplus W_{j-3})_{\lll 1} & 16 \le j \le 79, \end{cases}$$

where $X_{\lll n}$ indicates a circular left shift of the word X by n bit positions.

- Five working registers of size 32 bits A, B, C, D, E.
- A hash value H_i consisting of five 32-bit words ${H^0}_i$, ${H^1}_i$, ${H^2}_i, {H^3}_i, {H^4}_i$. In the beginning, the hash value holds the initial value H_0, which is replaced by a new hash value after the processing of each single message block. The final hash value $H_n$is equal to the output $h(x)$ of SHA-1.



The four SHA-1 stages have a similar structure but use different internal functions f_t and constants K_t, where $1 \leq t \leq 4$.
Every 20 rounds a new function and a new constant are being used, as shown in table.

Stage t	Round j	Constant K_t	Function f_t
1	0...19	$K_1 = 5A827999$	$f_1(B,C,D) = (B \wedge C) \vee (\bar{B} \wedge D)$
2	20...39	$K_2 = 6ED9EBA1$	$f_2(B,C,D) = B \oplus C \oplus D$
3	40...59	$K_3 = 8F1BBCDC$	$f_3(B,C,D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$
4	60...79	$K_4 = CA62C1D6$	$f_4(B,C,D) = B \oplus C \oplus D$

SHA-1 is **no longer used** as standard for hash functions since it has been broken by Google and declared insecure since 2015.
It has been replaced by SHA-3.

SHA-3

SHA-3 is internally different from the MD5 structure of SHA-1 and SHA-2.
Its original name is **Keccak Algorithm** and it is based on the idea of a **sponge construction**.
Sponge construction is based on a wide random function or random permutation, and allows inputting ("absorbing" in sponge terminology) any amount of data, and outputting ("squeezing") any amount of data, while acting as a pseudorandom function with regard to all previous inputs.

- **Absorbing phase :** input block x_i is read-in and processed.
- **Squeezing phase :** output is produced.

Output length:

- 224 (same resistance as 3DES when using birthday attack),
- 256
- 384 (same resistance as 192 AES when using birthday attack)
- 512

Keccak state size b, also called bus, is:

$$b = 25 \cdot 2^l \text{ where } l \in \{0, 1, 2, \ldots, 6\}$$
$$b = \{25, 50, 100, 200, 400, 800, 1600\}$$

SHA-3: only use b = 1600

Examples and Exercises

EXAM 2021 Jan 10th
Q3.1 Define and discuss the security requirements needed by a hash function inorder to be considered suitable for security purposes.

To be considered suitable for security purposes an hash function should be **non-invertible** and should generate an **unique fixed-length pseudorandom** output.
Moreover, an hash function should support some kind of resistances against attack:
Preimage resistance (or one-wayness) is the property that states that, given an output $y = h(x)$, it is computationally infeasible to find an input x such that $h(x) = y$.
Second preimage resistance (or weak collision resistance) states that given x_1 and $h(x_1)$ it is computationally infeasible to find an x_2 such that $h(x_2) = h(x_1)$. In this case the attacker can generate x_2.

Collision resistance property is the strongest resistance and states that it should be computationally infeasible to find any pairs of x_1, x_2 with $x_1 \neq x_2$such that $h(x_1) = h(x_2)$. In this case the attacker can generate both x_1 and x_2.

Q3.2 Are these functions good candidates for new cryptographic hash functions?

a. f(x) = XOR(x, r) || r
b. f(x) = SHA-3(XOR(x, r)) || r

where x is the message, r is a random number (same number of bits of x), XOR is the bitwise exclusive or operator, || means concatenation.Motivate your answer with respect to the security requirements presented in Q3.1.

Function a) is not: it is non-invertible, it is a simple XOR, and so is weak with any kind of resistance.
Function b) can be a good candidate, since it uses SHA-3 that is considered secure and generates an unique output. However the concatenation of the random number r at the and can give some information to the attacker, such as the length of the message and if the message is too short it can be easily discovered with some brute force.

Message Authentication Code (MAC)

Message Authentication (or **Data Origin Authentication**) is a property that a message has not been modified while in transit (**data integrity**) and that the receiving party can verify the source of the message.
A Message Authentication Code is also known as **Cryptographic Checksum** or **Keyed Hash Function**.
In terms of security functionality, MACs share some properties with digital signatures, since they also provide **message integrity** and **message authentication**.

RECALL
(Data) Integrity: information is not tampered while in transit. Crucial for ensuring that data has not been altered by a third party.
Message Authentication: the sender/creator of the message is authentic. Often includes integrity of the message.
Non Repudiation: the sender of a message cannot deny the creation of the message.

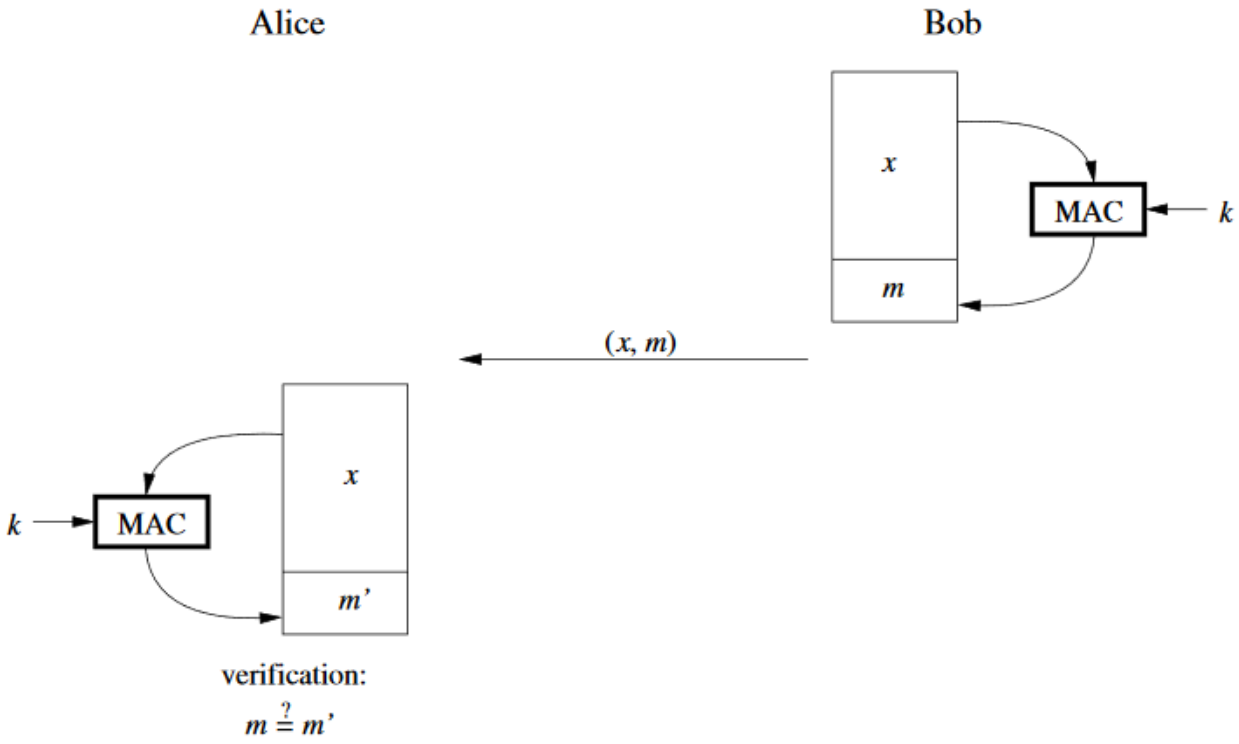
KEY POINTS

- Message authentication is a mechanism or service used to verify the integrity of a message. Message authentication assures that data received are exactly as sent by (i.e., contain no modification, insertion, deletion, or replay) and that the purported identity of the sender is valid.
- Symmetric encryption provides authentication among those who share the secret key.
- A message authentication code (MAC) is an algorithm that requires the use of a secret key. A MAC takes a variable-length message and a secret key as input and produces an authentication code. A recipient in possession of the secret key can generate an authentication code to verify the integrity of the message.
- One means of forming a MAC is to combine a cryptographic hash function in some fashion with a secret key.
- Another approach to constructing a MAC is to use a symmetric block cipher in such a way that it produces a fixed-length output for a variable-length input.

Message Authentication **does not always imply non repudiation** (which is offered by Digital Signature schemes).

MACs append an authentication tag to a message.

The difference between MACs and Digital Signatures is that MAC uses a **symmetric key** k both for generating the authentication tag m and verifying it.
A MAC is a function of the symmetric key k and the message x: $m = MAC_k(x)$.



Why to use MACs? Because doing so, Alice and Bob want to be assured that any manipulations of a message x in transit are detected.
Bob computes the MAC as a function of the message x and the shared key k. He sends both the message x and the calculated authentication tag m.
Alice, upon receiving the message and m can verify both.
The assumption is that the MAC computation will yield an incorrect result if the message x was altered in transit, providing so **message integrity**.
Alice can also assume that Bob itself sent the message since only the two parties share the secret key k, and this grants the **message authentication**.
If an attacker Trudy changes the message during the transit, he cannot recompute the MAC since he does not know the key k.

In practice, however, a **message x is often larger than the corresponding MAC**, while the output of a MAC computation is a fixed length authentication tag which is independent of the length of the input (similarity with Hash Functions).

Some important properties have to be respected.
(From "*Understanding Cryptography - Christof Paar*", cpt. 12.2

Properties of Message Authentication Codes

1. **Cryptographic checksum** A MAC generates a cryptographically secure authentication tag for a given message.
2. **Symmetric** MACs are based on secret symmetric keys. The signing and verifying parties must share a secret key.
3. **Arbitrary message size** MACs accept messages of arbitrary length.
4. **Fixed output length** MACs generate fixed-size authentication tags.
5. **Message integrity** MACs provide message integrity: Any manipulations of a message during transit will be detected by the receiver.
6. **Message authentication** The receiving party is assured of the origin of the message.
7. **No nonrepudiation** Since MACs are based on symmetric principles, they do not provide nonrepudiation.

Standing to the book *"Cryptography and Network Security - William Stallings",* cpt 12.1, in the context of communications across a network the following **attacks** can be identified.

1. **Disclosure:** Release of message contents to any person or process not possess-ing the appropriate cryptographic key.
2. **Traffic analysis**: Discovery of the pattern of traffic between parties. In a connection-oriented application, the frequency and duration of connection could be determined. In either a connection-oriented or connectionless environment, the number and length of messages between parties could be determined.
3. **Masquerade**: Insertion of messages into the network from a fraudulent source.This includes the creation of messages by an opponent that are purported to come from an authorized entity. Also included are fraudulent acknowledgments of message receipt or nonreceipt by someone other than the message recipient.
4. **Content modification**: Changes to the contents of a message, including insertion,deletion, transposition, and modification.
5. **Sequence modification**: Any modification to a sequence of messages between parties, including insertion, deletion, and reordering.
6. **Timing modification**: Delay or replay of messages. In a connection-oriented application, an entire session or sequence of messages could be a **replay** of some previous valid session, or individual messages in the sequence could be delayed or replayed. In a connectionless application, an individual message (e.g., datagram) could be delayed or replayed.
7. **Source repudiation**: Denial of transmission of message by source.
8. **Destination repudiation**: Denial of receipt of message by destination.

A verification algorithm for a MAC should return “accept” or “reject” based on key k, message x, $MAC_k(x)$.
Message space is extremely large.
MAC function is **not a 1-to-1 mapping**.
An adversary should not be able to construct (**forge**) a new legal (**valid**) pair $(x, MAC_k(x))$ even after seeing valid pairs from previous communication sessions.

ADVERSARIAL MODEL

Assumptions:

- the MAC function is known
- known valid past pairs: $(x_1, MAC_k(x_1))$, $(x_2, MAC_k(x_2))$, $(x_3, MAC_k(x_3))$, ...
- the adversary can request $MAC_k(x)$ for a given x

Goal: Find a new legal pair $(y, MAC_k(y))$ efficiently and with non negligible probabilty. The attack is successful even when y is meaningless (in general defined when a message is meaningful is not trivial and requires other rules).

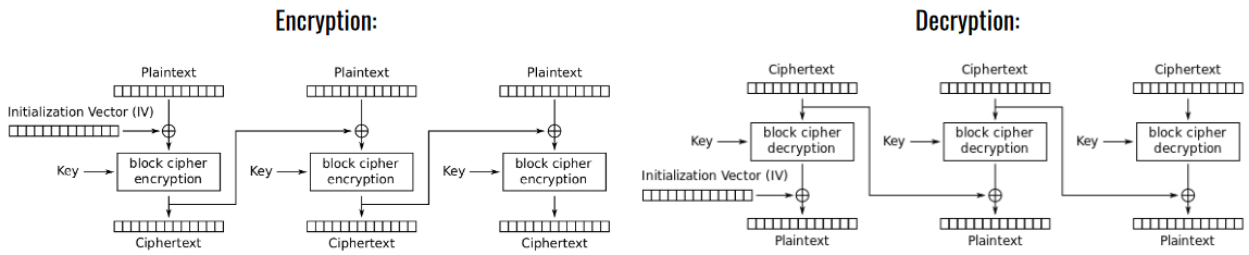
Implementation of Message Authentication Codes

Two ways for implementing MAC:

1. Using **Block Ciphers** (e.g. AES in CBC mode) : makes sense, cause we have a key. However it is very slow, because AES is slower than Hash Functions. Using a symmetric cipher is easy to know how to use the key, but the question is “How to get data integrity?”.
2. Using **Cryptographic Hash Functions**: more used in practice, cause this also helps with data integrity. However, here how to use the key?

MACs from Block Ciphers

An idea is to use the **CBC operation mode**.



- block y_i depends on y_{i-1}
- encryption is randomized using an IV
- encryption: no parallelization
- decryption: parallelizable

- if one bit flipped in x_i then all subsequent blocks are affected
- if one bit is flipped in y_{i-1} then xi is affected an unpredictable manner, while x_i in a predictable manner. This could be exploited by an attacker. Hence use CRC/etc.

The idea is the same: split the message in blocks and then apply the CBC mode.
Why to use an operation mode? To cope with long messages, since AES can only process a fixed amount of bytes.
We run a CBC mode using AES as a block cipher.

We **fix the value of the IV to zero**, that is something that is constant (so it is possible to write it in the implementation) and run the CBC and for each block of the message you will get a ciphertext block.
Why to use a constant IV? If the IV can be variable, the attacker can easily forge a new message x' similar to the previous message x but with few changes and then compare them maybe with a XOR of the different IVs, catching information about the MAC.
Run CBC for all the blocks, then trash all the other blocks and use **only the last block**: this is the **authentication tag**. Remember: we are not looking to recover the entire message from the authentication tag, we are not looking for confidentiality! We just want to have a computation that can process many blocks depending on the key.

Thus, this procedure requires two algorithms to be performed: **MAC Generation** and **MAC Verification**.

MAC Generation

- Divide the message x into blocks x_i
- Compute first iteration $y_1 = e_k(x_1 \oplus IV)$
- Compute $y_i = e_k(x_i \oplus y_{i-1})$ for the next blocks
- Final block is the MAC value: $m = MAC_k(x) = y_n$

MAC Verification

- Repeat MAC computation (m')
- Compare results: in case m' = m, the message is verified as correct
- In case m' ≠ m, the message and/or the MAC value m have been altered during transmission

Security of CBC-MAC of both fixed and variable length

FIXED LENGTH CBC-MAC
You have a message with fixed size blocks, maybe because you know that your application is exchanging messages of this fixed size, so the attacker can't shrink or reshape the message.
If e_k is a pseudo random factor, then the **fixed length** CBC MAC is resilient to forgery.

Proof. Assume CBC MAC can be forged efficiently. Transform the forging algorithm into an algorithm distinguishing e_k from a pseudo-random function efficiently.
By contradiction, if you can forge the authentication tag basically this means that you can break the block cipher.

VARIABLE LENGTH CBC-MAC
The **variable length** CBC MAC is insecure.

Proof. If an attacker knows correct message-tag pairs (x, t) and (x', t'), where $t, t'$are the message tags, and x has L blocks and x' has L'blocks, then he can generate a third longer message x'' whose tag will also be t':

$$x'' = x_1 \,||\, \ldots \,||\, x_L \,||\, (x'_1 \oplus t) \,||\, x'_2 \,||\, \ldots \,||\, x'_{L'}$$

XOR first block of x' with t and concatenate x with this modified x'. The resulting (x'', t') is a valid pair. This works because XOR operation with t "cancel out" the contribution from x.
So x'' has an authentication tag that is equal to t', the tag for x'.

** The notation $a \,||\, b$ means "Append b to a = put b after a".

How to improve CBC-MAC to make it more secure?

- **Input-length key separation** : generate a new key $k' = E_k(L)$ to use in CBC-MAC where L is the message length. Hence, messages with different lengths use different keys. This will prevent an attacker from reusing authentication tags on messages with a specific length for composing messages with a larger length.
- **Length-prepending** : include message length in the first block, i.e. $x\grave{\imath} = L \,||\, x$, where L is the message length. Problem: when processing a stream we may not know the length at the beginning. E.g. in a streaming service: you don't know when the live will end. What if the length is appended at the **end of the last block** instead? It is still insecure, cause with two messages a forgery like the one above for the *variable length CBC-MAC* is still possible with some changes.
- **Encrypt last block** : ECBC-MAC $E_{k2}(CBC - MAC_{k1}(x))$. Problem: we need to use two different keys.

MACs from Hash Functions

MAC is realized with cryptographic hash functions (e.g. SHA-1). In particular, when we combine the message and key and then perform hashing we call the process **keyed hashing**.
In this way it is possible to also achieve **data integrity**, by design of hash functions.
Any hash function can be used, combining it with the secret key.
** The notation $a \,||\, b$ means “Append b to a = put b after a”.

Idea: key is hashed together with the message, e.g.,

- **secret prefix MAC: $m = MAC_k(x) = h(k \,||\, x)$**
- **secret suffix MAC: $m = MAC_k(x) = h(x \,||\, k)$**

Remember that with the Merkle-Damgard construction an hash function can process messages with a **variable length**.

Security of MACs from Hash Functions

SECRET PREFIX MAC - ATTACK

$$m_x = h(k||x)$$

Given $(x = (x_1, x_2, \ldots, x_n), m_x)$, an attacker can easily forge $(x' = (x_1, x_2, \ldots, x_n, x_{n+1}), m'_x)$ without knowing the secret key as:

$$m'_x = h(x_{n+1})$$

Using $IV = m_x$ during Merkle-Damgård construction.

If we have a valid pair and the tag, an attacker can forge a new valid pair by adding a new block to the message and basically performing one more step in the Merkle-Damgard construction and so he gets the right authentication tag.

This approach is **not secure** and should not be used in practice.

SECRET SUFFIX MAC - ATTACK
The key is appended after the whole message (not after each block).

$$m_x = h(x||k)$$

Assume adversary can find a collision h(x) = h(x') then also h(x||k) = h(x'||k) = m_x

Notice that this attack is interesting even if it requires to find a collision as it lower the attack complexity with respect to a brute force attack wrt the key space:

- brute force attack: if |k|=128 then 2^{128} attempts to forge MAC
- collision attack: if output of h() is 160 bits then $2^{160/2=80}$ attempts to forge MAC

It is not possible to use the same attack as before.
A **brute force attack can be performed**, but it is not feasible if we use a large key.
However, since we have a hash function, it is possible to use the **birthday paradox** to find collisions, so we should enlarge the key. If the output of h() is 160 bits, then the attacker would need $2^{160/2} = 2^{80}$attempts to forge MAC.

HMAC

It is also called **Keyed Hashing MAC**. It is a mode of operation that can be used inside an hash algorithm, so it is **not an hash algorithm**. It receives in input a **key k** and an **hash function h** and computes the output.
Used in practice.

Idea: use 2 nested secret prefix MACs, e.g., h(k || h(k || x))

HMAC is an implementation of this idea with some additional details. It is extremely popular and used in several protocols.

HMAC is provable secure which means (informally speaking) that is secure if the hash function is secure.

In this way we **nest** a hash computation inside another hash computation.
If you use a secure hash function, then the HMAC is also secure.
If two messages collide in the inner hash, then they will collide for the whole hashing scheme, so HMAC suffers the **birthday paradox**.

HMAC vs BIRTHDAY PARADOX

- an attacker cannot generate MAC by himself with HMAC because he does not know the key k (he can still brute force k but then the number of attempts depends on the key space)
- to exploit a collision he needs to find it: he need to intercept (as it cannot generate them) on average, when output space is 160 bit, 2^{80} different HMACs using the same key k to have a collision probability of 0.5.
- hence, HMAC is vulnerable to a birthday attack (this is true for any hash function!), but performing it is quite impractical.

EXAMPLE Exam 2015 Feb 10th

Q1.1 - Describe what we mean by data integrity and discuss the use of keyed HMACs for guaranteeing the integrity of a file being transmitted over the network (no other guarantees requested).

Data Integrity is a property for which information is not altered by an unauthorized adversary while in transit in any step of its life-cycle.
One strategy is to enforce the standard **MAC** (Message Authentication Code). Remember that MAC consists in taking a message and generating a Message Auth Code using a shared key, and then in sending both the message and the MAC so that the recipient in possession of the key can generate again the MAC from the message and compare it to the one sent to verify authentication. The process of MAC generation can be done through Block Ciphers or through Hash Functions.
HMAC involves the concatenation of an hash function h such that the generated authentication code is equal to $h\,(k \,||\, h\,(k \,||\, x))$, where k is the key and x is the message.
In this case, data integrity is **granted by the use of hash functions**, so it is considered secure since hash functions are considered secure. *Why?* Because if collisions are very rare in the chosen hash function, then the sent HMAC will be specific for that message x and that used key k, so the generated HMAC will be **unique**: if the message x has been altered during transmission, then the HMAC generated by the recipient will be different from the received HMAC.
However, like the hash functions, HMAC is susceptible to the **birthday attack**: an attacker could change the message x to x' to find a collision in a way that then $h(k \,||\, x) = h(k \,||\, x')$. Doing so, the attacker grants to obtain the same HMAC also in the outer hash computation, since if the inner hash is equal, then the outer is equal, too. By the way, to do so the attacker must know the key k. If this situation happens, then **data integrity is not granted anymore,** since the sent

message will be equal to the received message, but the receiver can't notice the difference since his computed HMAC is equal to the received one.

Q1.2 - Suppose you are requested to ensure the integrity of a file but you are only allowed to use AES (and a symmetric key): what can it be done?

To guarantee Data Integrity using AES it can be used the **CBC operation mode** to generate a MAC. The CBC mode is needed since a standard AES can only cope with messages of a fixed amount of bytes. So, run an AES with CBC mode.
In this case, the IV must be full of zeroes. If IV is not constant, then an attacker could generate a message x' similar to the message x.
To generate the MAC only the last block is used. We don't care about the full message, we only need an authentication code. Since in CBC mode the last block depends from the operations done on all the blocks of the message, this can be good to be used as MAC: if a byte changes in block i, then this change is propagated into all the subsequent blocks.
That is, the MAC will be **equal to the last block** of the message encrypted in AES CBC mode.
So the message x is sent with the MAC.
For the **MAC verification**, the receiver computes the MAC code applying AES **encryption** to the message x and comparing his obtained MAC with the received one.
Of course, both sender and receiver must have a shared key k to perform AES encryption.
This method is strong and resists the birthday attack because it does not involve hash functions, but it is way too slow since AES encryption is slower than hash computing.

Authentication Encryption (AE)

Authenticated encryption (AE) and Authenticated Encryption with Associated Data (AEAD) are forms of encryption which simultaneously assure the confidentiality and authenticity of data.

Additionally, authenticated encryption can provide security against chosen ciphertext attack. In these attacks, an adversary attempts to gain an advantage against a cryptosystem (e.g., information about the secret decryption key) by submitting carefully chosen ciphertexts to some "decryption oracle" and analyzing the decrypted results. Authenticated encryption schemes can recognize improperly-constructed ciphertexts and refuse to decrypt them.

We want to add **confidentiality**. Maybe sometimes you want to send a payload that is confidential, you want to know who is the sender of the message, the payload is not corrupted and you want to use public information for sending the message.

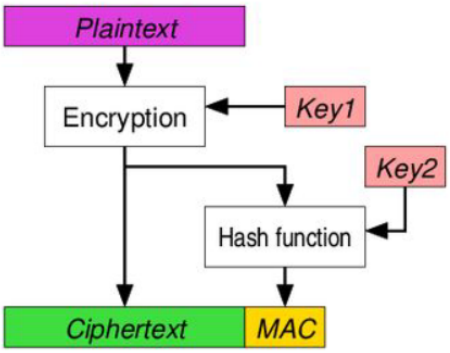
We don't want the attacker to build ciphertext to send to the user (**chosen ciphertext attack**). One way to prevent this is, given a ciphertext, after decryption is important to be sure that the person who sent the message actually **owns** the key.

Different ways of implementing this:

1. Encrypt-then-MAC (EtM)
2. Encrypt-and-MAC (E&M)
3. MAC-then-Encrypt (MtE)

Encrypt-then-MAC (EtM)

The plaintext is first encrypted, then a MAC is produced based on the resulting ciphertext. The ciphertext and its MAC are sent together.

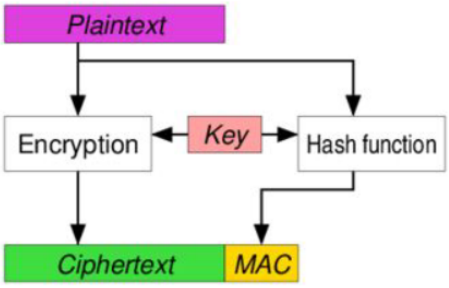


Used in, e.g., IPsec. This is the only approach which can reach the highest definition of security in AE accorting to ISO/IEC 19772:2009, but this can only be achieved when the MAC used is "strongly unforgeable".

Various EtM ciphersuites exist for SSHv2 and TLS.

Encrypt-and-MAC (E&M)

A MAC is produced based on the plaintext, and the plaintext is encrypted without the MAC. The plaintext's MAC and the ciphertext are sent together.

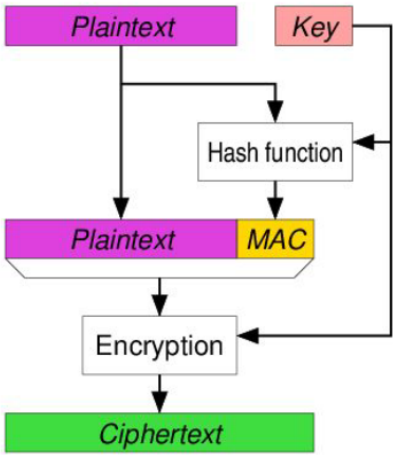


Used in, e.g., SSH. Even though the E&M approach has not been proved to be strongly unforgeable in itself, it is possible to apply some minor modifications to SSH to make it strongly unforgeable despite the approach.

MAC-then-Encrypt (MtE)

A MAC is produced based on the plaintext, then the plaintext and MAC are together encrypted to produce a ciphertext based on both. The ciphertext (containing an encrypted MAC) is sent.

Used in, e.g., SSL/TLS. Even though the MtE approach has not been proven to be strongly unforgeable in itself, the SSL/TLS implementation has been proven to be strongly unforgeable due to the encoding used alongside the MtE mechanism.



Examples and Exercises

EXERCISE 1 - What are the benefits of introducing keyed hashing?
The MAC is a fixed length code generated after processing a message x using a key k. It is used to guarantee **data origin authentication** (if the key k is not somehow intercepted) and **data integrity**, because the receiver calculates again the MAC on the message x using the key k and compares it to the received MAC.
MAC can be calculated using a CBC block cipher or using hash functions, and in this case it is called **keyed hashing**.
The benefits are that differently from a simple hashing, a keyed hashing can guarantee also data origin authentication other than data integrity, because it is supposed that the key is only known by the two parties.
Comparing keyed hashing with MAC using CBC the benefits are that keyed hashing is faster than MAC with CBC mode.
Keyed hashing can be implemented using two strategies: **secret suffix** $[m = h(x||k)]$ or **secret prefix** $[m = h(k||h)]$**,** both with their own weaknesses.

EXERCISE 2 - Can the non-repudiation be guaranteed by keyed hashing?
NO, non-repudiation can't be granted using keyed hashing since the key k used for generating the MAC is a **symmetric key**, and if it is somehow intercepted anybody can use it.
Even if the secret key is securely guarded and nobody else knows it except for the two parties A and B that uses it, a message can be intercepted and can be sent **again** in a second moment by the attacker, and Bob can tell Alice that he never sent the message twice.
So data origin authentication is still valid, since the message is truly originated by Bob, but nobody can be sure that Bob itself sent the message two times.
To guarantee non repudiation some techniques can be used, such as the use of **timestamps**, a **challenge** made with nonces or an asymmetric key scheme, maybe with **digital signatures**.

EXERCISE 3 - Is the function $h(x) = x \, mod \, 2^{256}$ a cryptographic hash function?
NO, it isn't. A hash function has to be non-invertible and should also generate an unique result. If we have two numbers x and $x + 2^{256}$ we can easily generate a collision, and this makes our function not so secure.
Moreover, this function is not resistant to the main security requirements of hash functions:

- **Preimage resistance (one wayness):** given an output $y = h(x)$it should be computationally infeasible to find any input x such that $h(x) = y$.
  Even if we cannot find the exact value of x given y, we could restrict the domain of values for x to exploit. If we know that the message is $< 2^{256}$then the set of the possible inputs only contains a single element!
- **Second preimage resistance:** given $x_1, h(x_1)$it should be computationally infeasible to find any x_2 such that $h(x_1) = h(x_2)$.
- **Collision resistance**: it should be computationally infeasible to find pairs (x_1, x_2) such that $h(x_1) = h(x_2)$ with $x_1 \neq x_2$.
  In this function, as said before, we can easily find collisions picking x_1 and any number $x_2 = x_1 + 2^{256}$.

Digital Signatures

KEY POINTS

- A digital signature is an authentication mechanism that enables the creator of a message to attach a code that acts as a signature. Typically the signature is formed by taking the hash of the message and encrypting the message with the creator's private key. The signature guarantees the source and integrity of the message.
- The digital signature standard (DSS) is an NIST standard that uses the secure hash algorithm (SHA).

- Alice orders a pink car from the car salesmen Bob
- After seeing the pink car, Alice states that she has never ordered it:
- How can Bob prove towards a judge that Alice has ordered a pink car? (And that he did not fabricate the order himself)
  - This is security property is called non repudiation
  - Symmetric cryptography fails because both Alice and Bob can be malicious: hence MAC cannot be used when non repudiation is required
  - Can be achieved with public-key cryptography

We need a **public-key scheme** to solve this kind of problem in order to guarantee **non repudiation**.
Again, we have a **generation algorithm** for generating the signature and a **verification algorithm**.

Basic Digital Signature Protocol

Alice		Bob
		generate $k_{pr,B}$, $k_{pub,B}$
	$\xleftarrow{k_{pub,B}}$	publish public key
		sign message: $s = \text{sig}_{k_{pr}}(x)$
	$\xleftarrow{(x,s)}$	send message + signature
verify signature: $\text{ver}_{k_{pr,B}}(x,s) = \text{true/false}$		

It is basically the opposite of public key cryptography: here we are **inverting the role** of the public and private key.
In, for example, RSA to gain **confidentiality** we use the public key for encryption and the private key for decryption.
Here we do the opposite: who wants to generate something uses the private key and who wants to verify takes the output of the message encrypted with the private key and decrypts it using the public key.

Digital signature has many **features**:

- For a given message x, a digital signature is appended to the message.
- Only the person with the private key should be able to generate the signature.
- The signature must change for every document.
- The signature is realized as a function with the message x and the private key as input.
- The public key and the message x are the inputs to the verification function.

Security Services from Digital Signatures

- **INTEGRITY** : ensures that a message has not been modified in transit
- **MESSAGE AUTHENTICATION** : ensures that the sender of a message is authentic. An alternative term is Data Origin Authentication
- **NON REPUDIATION :** ensures that the sender of a message can not deny the creation of the message.

Attacks and Forgery

A is the user of a DS, C is the attacker.

- **Key-only attack:** C only knows A's public key.
- **Known message attack:** C is given access to a set of messages and their signatures.
- **Generic chosen message attack:** C chooses a list of messages before attempting to breaks A's signature scheme, independent of A's public key. C then obtains from A valid signatures for the chosen messages. The attack is generic, because it does not depend on A's public key; the same attack is used against everyone.
- **Directed chosen message attack:** Similar to the generic attack, except that the list of messages to be signed is chosen after C knows A's public key but before any signatures are seen.
- **Adaptive chosen message attack:** C is allowed to use A as an "oracle." This means the A may request signatures of messages that depend on previously obtained message–signature pairs.

- **Total break:** C determines A's private key.
- **Universal forgery:** C finds an efficient signing algorithm that provides an equivalent way of constructing signatures on arbitrary messages.
- **Selective forgery:** C forges a signature for a particular message chosen by C.
- **Existential forgery:** C forges a signature for at least one message. C has no control over the message. Consequently, this forgery may only be a minor nuisance to A.

Forgery is the ability to create a pair consisting of a message x and a signature (or MAC) s that is valid for x, where x has not been signed in the past by the legitimate signer.
For MAC, is the ability of the attacker to generate a valid pair, so to generate an authentication tag with respect to a message that was not processed by an user owning the key.
Here in the DS we have three parties: the **user** owning the private key, the **attacker** that will try to forge a new signature and the **challenger** that asks the attacker to forge a signature for a specific message (not any kind of message) and his role is to make the life of the attacker harder .
Three types of forgery: **Universal Forgery, Selective Forgery, Existential Forgery**.

Universal Forgery

- adversary creates a valid signature s for any given message x (chosen by the adversary or by a challenger)

- it is the strongest ability in forging and it implies the other types of forgery

The attacker is very powerful, and if the attacker can do this, then can do anything else.

Selective Forgery

- adversary creates a message/signature pair (x, s) where x has been chosen by the challenger prior to the attack
- x may be chosen to have interesting mathematical properties with respect to the signature algorithm; however, in selective forgery, x must be fixed before the start of the attack
- the ability to successfully conduct a selective forgery attack implies the ability to successfully conduct an existential forgery attack

Here the challenger choses a message to give to the attacker, so the attacker can't pick any message.
The challenger can choose a message with a specific structure **at the beginning** but then he can't choose the successive messages.
Selective Forgery implies Existential Forgery.

Existential Forgery

- adversary creates at least one message/signature pair (x,s), where s was not produced by the legitimate signer.
- adversary can choose x freely: x need not have any particular meaning
- existential forgery is essentially the weakest adversarial goal
- creating an existential forgery is easier than a selective forgery, because the attacker may select x for which a forgery can easily be created, whereas in the case of a selective forgery, the challenger can ask for the signature of a "difficult" message.
- therefore the strongest schemes are those which are "existentially unforgeable"

Implementing a Digital Signature scheme based on a PK scheme

We may consider two approaches:

- sign a message x using the public key: $(x, s = E_{pub}(x))$
  - problem: public keys are public, then everybody can forge a signature
  - remark: sending only $s = E_{pub}(x)$ without the message x provides confidentiality but not authentication
- sign a message x using the (my) private key: $(x, s = E_{priv}(x))$
  - only the owner of the private key can generate the signature s
  - problem: this scheme is still affected by existential forgery (see next slides)
  - remark: sending only $s = E_{priv}(x)$ provides authentication but no confidentiality

Why **no confidentiality** in this last approach? I generate something that anybody can decrypt with the public key.
The two approaches are **complementary**, and this is a problem, cause we want to avoid to merge together the two approaches.

We still have a problem: existential forgery, i.e. when the attacker is able to generate a signature for a message that he chooses, maybe even with a nonsense message.

Digital Signature schemes on RSA

Assume we are using RSA as PK scheme for implementing the digital signature scheme:

1. To generate the private and public key: use the same key generation as RSA encryption
2. To generate the signature: "encrypt" the message x with the private key

   $$s = sig_{Kpriv}(x) = x^d \bmod n$$

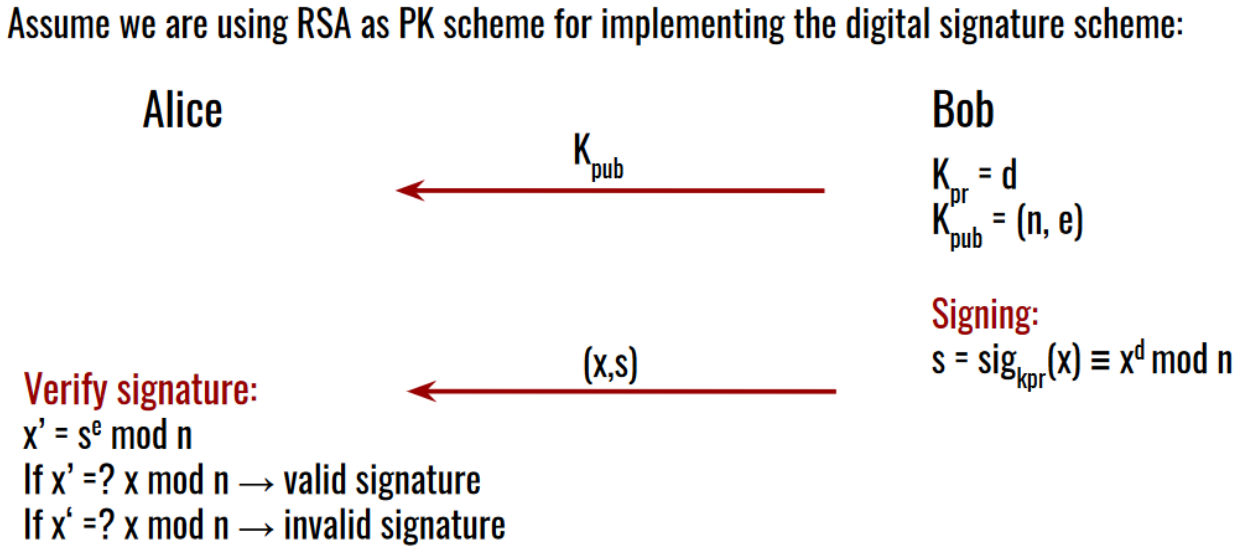
   then append s to x by sending (x, s)
3. To verify the signature: "decrypt" the signature with the public key

   $$x' = ver_{Kpub}(s) = s^e \bmod n$$

   If x=x', the signature is valid

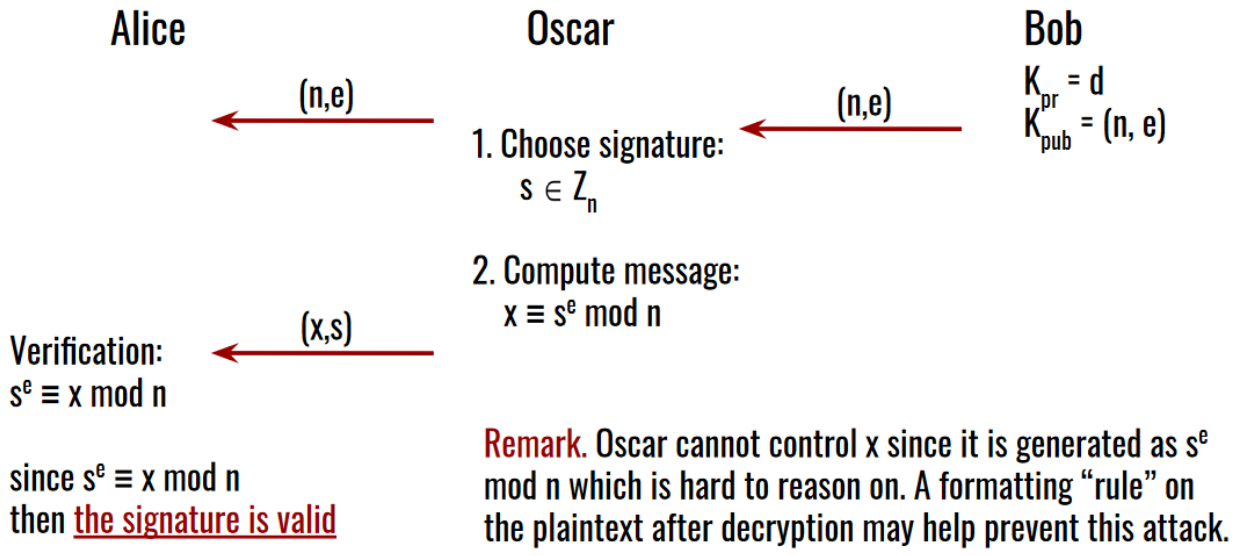
Step 1 for generating public and private keys is the same as done in RSA.
Step 2 consists in just performing encryption, **but** instead of using the public key from RSA, **use the private key**. The output of this will be the signature s.
How large is s? A common approach is to **encrypt the hash of x** $h(x)$ instead of the full message x.

Step 3 is like a decryption in RSA **but** performed **with the public key** instead of with the private.



It is the **same as RSA but with swapping** the role of public and private keys.

PROBLEM: using **existential forgery** an attacker Oscar can sign in. If he intercepts the key exchange from Bob with the exponent e, he can compute the message x by himself.



Oscar is fooling Alice generating a signature accepted by Alice, but he **cannot control the content of the message x**, so the message x does not have any meaning.
This is an existential forgery attack because the content of the message x does not have any particular meaning.

Why perform such an attack? Maybe no verification for the content of the message is needed, the message seems to be sent by Bob but really it is not, but in some context this may be sufficient for being accepted.We need a way to define when x is valid!
PROBLEM : ciphertext malleability.

Several PK schemes, such as RSA, are malleable, i.e., an attacker can transform a ciphertext into another ciphertext which decrypts to a related plaintext without knowing the private key. For instance in RSA "textbook", an attacker can exploit the property that:

$$(x_1 \cdot x_2)^e \bmod n \equiv x_1^e \cdot x_2^e \bmod n$$

Similar problem in Elgamal (see slides on asymmetric encryption).

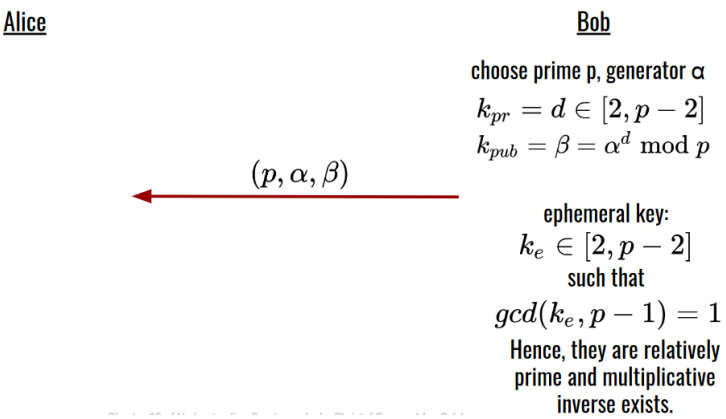
As for encryption, textbook implementations of PK scheme are not safe for digital signature mechanism. Always use standards that define proper rules (e.g., padding).

The attacker may be able to manipulate the ciphertext in some way such that the plaintext could be, for instance, multiplied by two.
The concept is that the plaintext will change.

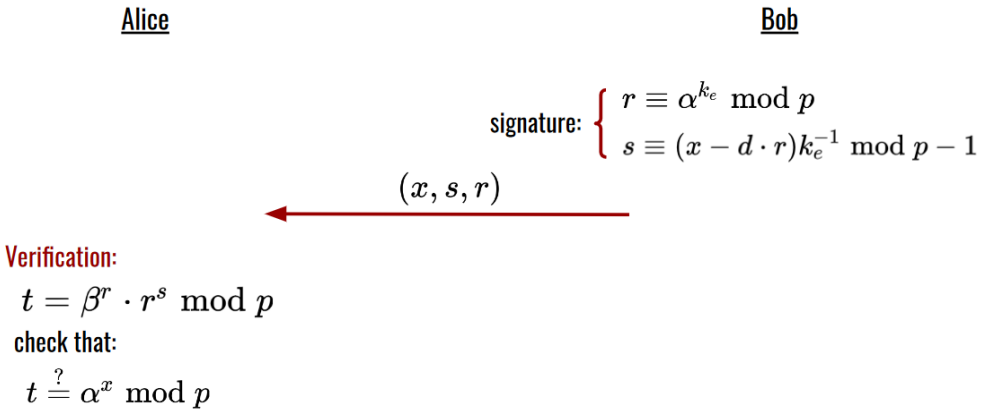
Elgamal Signature Scheme

Some standards are used to secure Digital Signatures: **PKCS#1** (using RSA to sign the hash of the message and padding with formatting rules) **Elgamal Signature Scheme**, **Digital Signature Standard (DSS)**.

Focus on **Elgamal Signature Scheme**, since DSS is a sort of redefinement of the first one.
The idea is the same, but with some differences.
Bob chooses the private key d and derives the public key β (where α is a generator).
Now is Bob choosing the ephemeral key, no more Alice, because now is Bob that wants to perform the encryption, and so he needs to compute the ephemeral key.



So Bob sends the public key that is (p, α, β) and then sends the signature that is the triple (x, s, r), where r is calculated as below.



Alice wants to verify the computation.
This is proved to work using Fermat's Little Theorem.

ElGamal DS recap - the scheme can be divided into three parts: **generation, signing, verification**.

Generation: Bob chooses a prime number p of 1024 bits such that the Discrete Logarithm in $Z_p{}^*$is hard to find and the only way to get the private key k is by computing the DL of p in base g.
Find a generator g of $Z_p{}^*$an integer for which the powers are congruent with the numbers coprime with p.
Pick randomly an x in [2, p-2] and compute $y = g^x mod\ p$.
The public key is $k_p = (p, g, y)$ and the private key is $k_s = x$.

Signing: if we want to sign a message M we define a new pair of keys (r, k) where r is public and k is private. Then start by computing the digest $m = h(M)$.
Pick k in [1, p-2] coprime to p-1 at random.
Compute $r = g^k mod\ p$ and $s = (m - rx)k^{-1} mod(p-1)$. If s is 0, then we need to restart.

The output signature will be (r, s).

Verification: compute $m = H(M)$.
We accept only if $0 < r < p\ OR\ 0\ <\ s\ < p-1\ OR\ (y^r \times r^s = g^m)\ mod\ p$
Since we have that $s = (m - rx)k^{-1} mod (p-1)$, we have that $sk\ +\ rx\ =\ m\ mod (p-1)$ disappears.
Now we know that $r = g^k mod\ p$ so if we power left and right operands for s we have $r^s = g^{ks} mod\ p$-
We also know that $y = g^x$ and powering for r we have $y^r = g^{xr} mod\ p$.
If we multiply we obtain $y^r \times r^s = g^{ks+xr} mod\ p\ =\ g^m\ mod\ p$.

Digital Signature Algorithm

It is based on the Elgamal Signature scheme and SHA-1, but other hash functions can be used.
Signature is only 320 bits long

Key generation of DSA:

1. Generate a prime p with $2^{1023} < p < 2^{1024}$
2. Find a prime divisor q of p-1 with $2^{159} < q < 2^{160}$
3. Find an integer α with ord(α) = q
4. Choose a random integer d with 0 < d < q
5. Compute $\beta \equiv \alpha^d \bmod p$

The keys are:

$k_{pub} = (p, q, \alpha, \beta)$

$k_{pr} = (d)$

DSA signature generation:

Given: message x, private key d and public key (p, q, α, β)

1.Choose an integer as random ephemeral key k_E with $0 < k_E < q$

2.Compute $r \equiv (\alpha^{kE} \bmod p) \bmod q$

3.Computes $s \equiv (SHA(x)+d \cdot r)\ k_E^{-1} \bmod q$

The signature consists of (r,s)

SHA denotes the hash function SHA-1 which computes a 160-bit fingerprint of message x.
Other hash functions can be used in place of SHA-1.

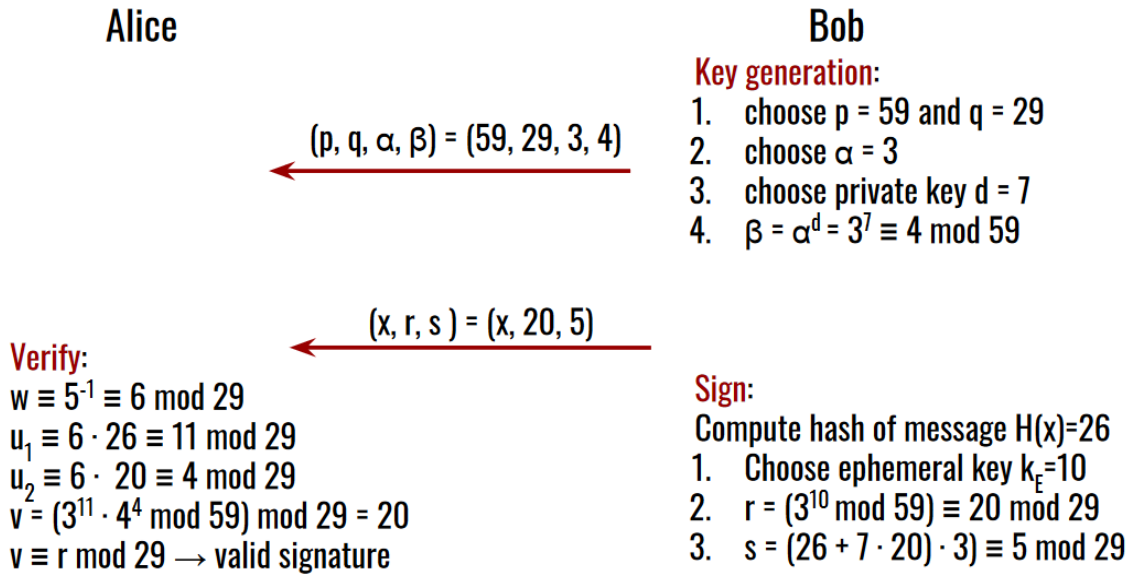
DSA signature verification

Given: message x, signature s and public key (p,q,α,β)

1. Compute auxiliary value $w \equiv s^{-1} \bmod q$
2. Compute auxiliary value $u_1 \equiv w \cdot SHA(x) \bmod q$
3. Compute auxiliary value $u_2 \equiv w \cdot r \bmod q$
4. Compute $v \equiv (\alpha^{u1} \cdot \beta^{u2} \bmod p) \bmod q$

If $v \not\equiv r \bmod q \rightarrow$ signature is valid

If $v \equiv r \bmod q \rightarrow$ signature is invalid



Examples and Exercises

EXERCISE 1 - Evaluate the truth of the following assertions

- A digital signature is obtained by encrypting a message digest by the public key of the signer → **FALSE**. A digital signature is obtained by encrypting a message digest with the **private key** of the signer. If we encrypt with the public key, then anybody is able to compute it and thus force the signatures of even non signed messages.
- ElGamal signature uses a temporary pair of public/private keys →**FALSE**, although Elgamal uses a different ephemeral key for each message, it can reuse the same public/private key for different messages since the ephemeral key plays the role of randomizer. Hence the keys are said “per-user”.
- DSS (DSA) signature uses a temporary pair of public/private keys → **FALSE**, for a similar argument as in ElGamal. In DSA (DSS), keys are “per-user”. An ephemeral random value is used to randomize the signing process of a message.

Authentication I

- Authentication: human vs computers
- Authentication through passwords
  - pitfalls and attacks
  - salted passwords, Lamport's Hash
- Protocols for authentication:
  - Two entities share a secret symmetric key
    - challenge-response based, timestamp based, attacks
  - Entities share a secret with a trusted entity
    - authentication server, Needham–Schroeder symmetric protocol
    - Kerberos
  - Entities have public/private keys
    - Needham–Schroeder public-key protocol
    - PKI, X.509

Authentication is the process of reliably verifying the identity of someone or something.
So both **humans** and **computers** can be authenticated.

Authentication of people

Different approaches can be combined to obtain a **multi-factor authentication**:

- **What you know**: passwords or secret questions (as for recovery procedures). Some problems are that those things are in general easily guessable by an attacker cause they are not truly random and they can be captured with specific softwares like trojans, fake logins, keyloggers or using social engineering techniques.
- **What you have**: authentication tokens can contain secret hidden keys that can directly communicate with a system for carrying out a specific protocol or generate a One Time Password. Bank token gadgets or smartphones can play this role. Some standards for OTP are HMAC-based One Time Password (HOTP) or Time-based One Time Password (TOTP).
- **Who you are**: mainly based on biometrics such as fingerprint, retina examination, voice recognition.
- **Where you are:** use of GPS to determine the location. Some websites or applications will often warn you about logins from unexpected locations.

Strong Customer Authentication (SCA) is a requirement of the EU Revised Directive on Payment Services (PSD2).
One way to perform SCA is **3-D Secure**, where three domains interact using the protocol: the merchant, the issuer and the interoperability domain.
The basic concept of the protocol is to tie the financial authorization process with online authentication: one common way is to redirect the user to a page (from the Bank) and ask for a

password that is tied to the current card. More recently, OTP via SMS.
The major concern is that **the user should understand** that the page (or iframe) is the right one.

HMAC-based One-Time Password algorithm (HOTP) [RFC4266]

Idea: compute a value using HMAC providing two inputs (secret K, counter C), which are known to both parties. The value must d digits, e.g., d=8. After each attempt, the counter is incremented.

$$value = HOTP(K, C) \bmod 10^d$$
$$HOTP(K, C) = truncate(HMAC(K, C))$$

Problem. The counter must been synchronized

Possible (partial) solution. The server will compute W values (e.g., W=100), considering many increasing value of the counter C and will check all of them. Hence, there is "window" for re-resynchronization in each attempt. However, if the client increments the counter too much (e.g., a kid pressing the button on the HTOP device), then it will not work even using a "window".

Time-based One-Time Password algorithm (TOTP) [RFC6238]

Idea: use a time-based value instead a synchronized counter in HOTP.

$$TOTP(K, C) = HTOP(K, C_T)$$

where:

$$C_T = \left\lfloor \frac{T - T_0}{T_X} \right\rfloor$$

Check this Python implementation!

- T is the current unix time
- T_0 is an initial time (unix epoch)
- T_X is the length of window duration (e.g., 30 seconds)

To make the process easier, the server will check will often consider T, T+1, T-1.

Authentication through Passwords

Some precautions must be taken:

- **Robustness** : passwords chosen by humans are weak as they are short and not

random, thus they can be guessed by an attacker.

- **Never send password in clear** : attacker can sniff the password when sent in clear, hence a authentication password should be different each time to prevent replay attacks.
- **Store passwords securely** : if the password is stored somewhere then it is crucial to store it safely, e.g., password manager should keep password encrypted.

Many operating systems store passwords for local users in local files. E.g. Linux passwords are stored in /etc/shadow. This file should contain an hash of the passwords, so that even if the file is leaked, passwords are unreadable from the attacker.

Against password a common attack is the **Dictionary Attack** and this can be performed **online** (but the throughput can be low, since some web apps make the authentication process slow) or **offline** done on a local machine, where the throughput can be very high.

Another problem can be that if an attacker gets a **dump of the database of hashed passwords**, it can easily detect when two users have the same password (the hash will be the same) and use this information for his purposes.

Salted Passwords

The idea is to use a randomizer for each user to “salt” the password:

$$h = hash(password \,||\, salt)$$

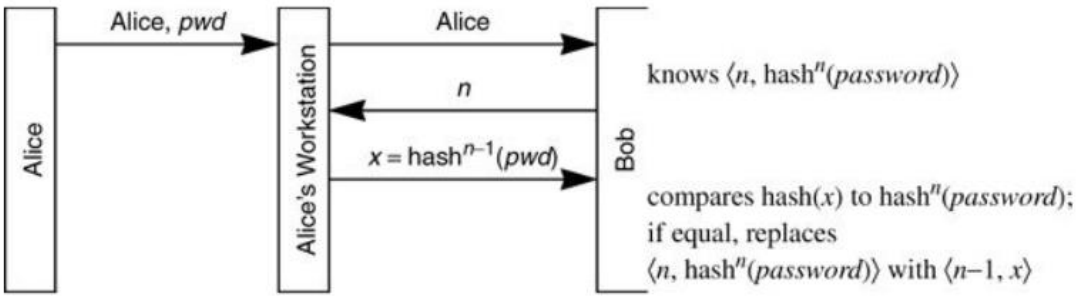
The **salt can be stored in clear** near the hash of the password. **Salt is not secret**, but is chosen randomly. Its role is to make the life of an attacker harder by requiring to perform the authentication procedure from scratch for each salt / for each user.
So if two users have the same password, the salt will make the hashes different: in this way precomputed tables of hashes are useless.

Lamport’s Hash

Useful to cope with **remote authentication**: if Alice wants to authenticate on a remote server using a client machine for computing the hash to send to the server, then the hash can be **intercepted** and reused later or being cracked by an attacker to get Alice’s password.

The idea on Lamport’s Hash is to avoid offline guessing when sending an hash through network:

- Bob stores $(n, hash^n(password))$ and sends to Alice n when auth request
- Alice computes and sends $hash^{n-1}(password)$
- Bob checks it comparing $hash^n(password)$ with $hash(hash^{n-1}(password))$, then updates entry to $(n-1, hash^{n-1}(password))$.



Remark. if hashing is incremented instead of decremented then attack is trial

Alice decrements instead of incrementing because if she increments, then an attacker can steal the hashed password and then send it "incremented hash", while doing so, if the attacker steals the hashed password then he can not "perform one less hash operation", cause he doesn't know the password (of course).

A **salt** can be added to Lamport's Hash:

$$(n, salt, hash^n(password \,||\, salt))$$

This makes it more secure to use the same password in different systems for the reasons specified above.

Some **attacks** to the Lamport's Hash can be:

- Bob is not authenticated to Alice: Man-in-the-Middle (MITM) attack is possible
  Small n attack:
  - Bob stores (n, hash^n(password))
  - Attacker impersonates Bob with Alice sending n' with n' << n
  - Alice provides hash$^{n'}$(password)
  - Attacker can authenticate (n - n') times since it just needs to perform hashing starting from hash$^{n'}$(password)

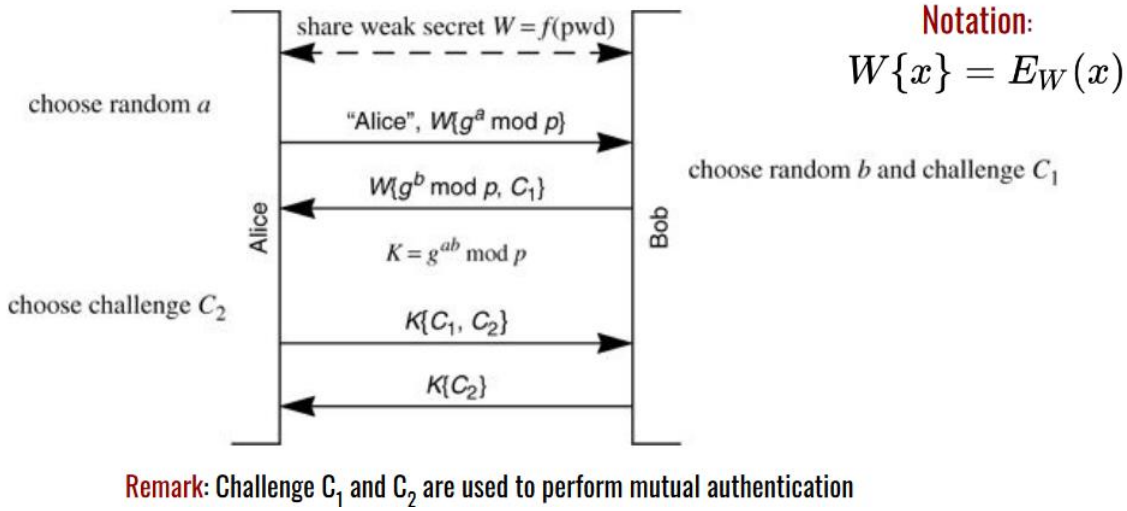
  Mitigation: Alice should keep track of n from Bob

- Machine of Alice cannot be dumb as it has to perform hashing. A solution ("human and paper" approach): hashes are precomputed and given to Alice, which then uses each of the hash only once for authentication. This is known as S/Key, standardized in RFC 1938 "A one-time password system"

EKE: Encrypted Key Exchange

Alice and Bob share a secret which is "weak", they want to build a "secure" secret key and the attacker should not gain any benefit from observing messages exchanged during this process.

The idea behind EKE is like Diffie-Hellman Key Exchange where numbers are encrypted using the weak secret W.



Remark: Challenge C_1 and C_2 are used to perform mutual authentication

Exchanged messages appear as random numbers. There is no way of performing an offline attack since a and b changes every time. However online attacks are still possible.
Neither replay attacks are possible for the same reason.
Given W, an attacker can authenticate but it is computationally hard to get W by looking at the messages.

To **recap:**

- A and B agree on a **weak secret** W.
- Alice chooses a random a and sends to Bob : $W(g^a mod\ p)$
- Bob is able to decrypt cause he knows W. Then chooses a random b and a challenge C_1 and replies to Alice with $W(g^b mod\ p, C_1)$.
  Then Bob computes the key as DH protocol: $K = g^{ab} mod\ p$
- Alice decrypts the message, so she computes the key $k = g^{ab} mod\ p$ and also reads C_1. Then she challenges Bob with C_2 sending $K(C_1, C_2)$.
- Bob can decrypt with the key, so he can read $C_2$and answer to Alice with $K(C_2)$.

This solution **solves the MITM attack weakness of DH**.
EKE is also **strong to replay attacks** since a is changed every time, and also to **dictionary attacks** even if the password W is weak, cause the choice of a random a doesn't allow the attacker to compute g^a.
The **authentication is strong** and the attacker can act in place of Alice only if he know the password.

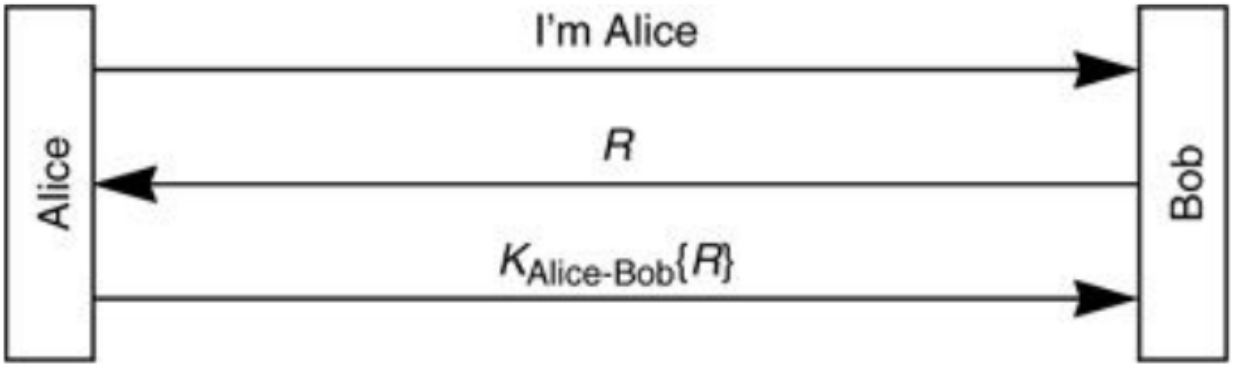
Variants of this protocol can be **SPEKE (Simple Password Exponential Key Exchange)** or **PDM (Password Derived Moduli).**

Authentication II

Different techniques to perform authentication:

- **Timestamps** : prevent attackers from reusing old messages; require clock synchronization among entities.
- **Nonce (or Challenge)** : random numbers, never re-used (Number used only ONCE), used to “challenge” someone to encrypt/decrypt something. In some protocols, nonce could be not random but a sequence number.

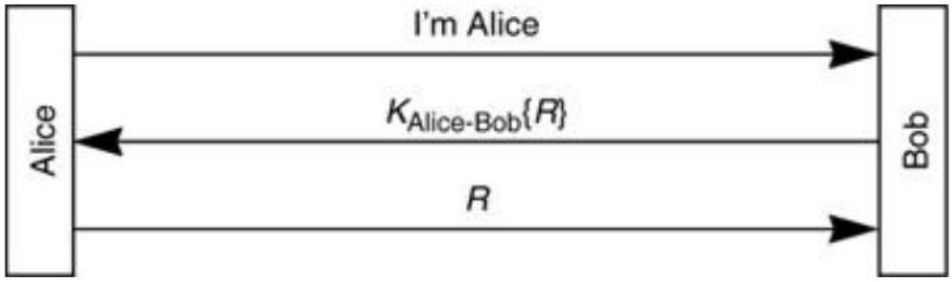
CHALLENGE-RESPONSE BASED ON SHARED SECRET - SCHEME 1



- R is a **nonce** with a **limited lifetime**
- The encryption function can be an hash function
- Bob authenticates Alice, but **not the opposite**, hence it is **not mutual authentication**
- If K_{AB} is derived from a password, then an attacker can still guess the password
- Offline password guessing: a valid pair is known to the attacker
- If Bob (e.g. the server) is compromised, then Alice is in Trouble
- Still better than sending password in clear (or hash of a password)

Problem: attacker can impersonate Bob and challenge Alice with R.

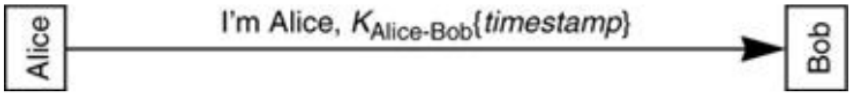
CHALLENGE-RESPONSE BASED ON SHARED SECRET - SCHEME 2



- Similar as before, but requires that the encryption is reversible (decryption), so no hash functions can be used.

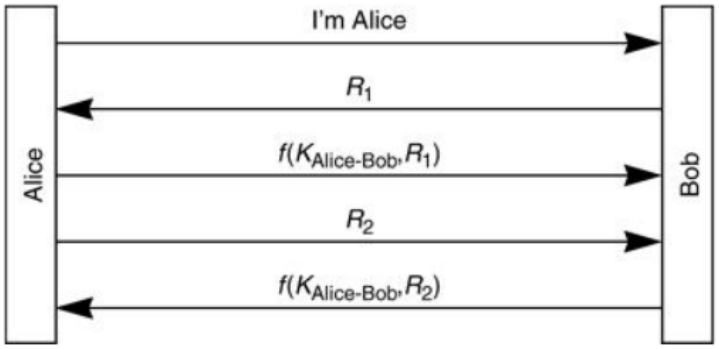
Problem: Alice is not challenging Bob: she cannot be sure that B is not just using old messages.

Timestamps to limit replay attacks

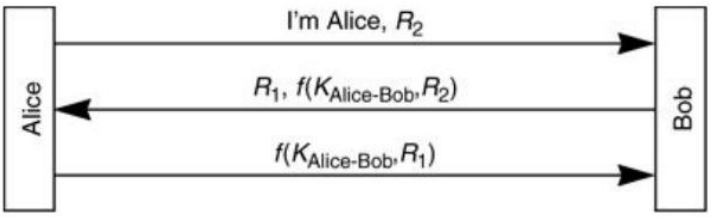


- Bob and Alice have reasonably synchronized clocks (can be a weakness: we need a protocol to set securely the correct time)
- Alice encrypts the current time, Bob decrypts the result and makes sure the result is acceptable (i.e., within an acceptable clock skew)
- efficient, no intermediate states
- if Bob remembers timestamps until they expire, then no replaying attacks
- if multiple servers with same secret K, since timestamp in not tied to Bob then Alice could send K{Bob|timestamp} to prevent problems
- no mutual authentication

Mutual Authentication



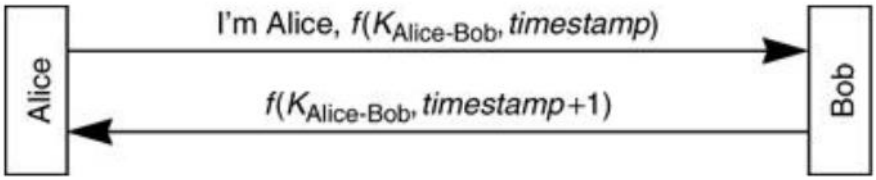
The number of messages **can be reduced!**
Alice and Bob should **keep track of nonce** over time to **avoid replay attacks**.



Fewer messages, more efficient.
However it is **weak to a reflection attack:** Trudy can open two sessions, sending R_1 in the second one pretending to be Alice, so he can compute the data needed.

Allso to **offline password guessing** on the side of Alice.

HOW TO PREVENT REFLECTION ATTACKS?
Use **different keys**, so cryptographic operations will be different.
Use different challenges from the initiator and the responder, e.g. using even and odd numbers.
The **Reflection Attack**, is an attack that aims to construct an extra session useful just for the adversary. The second session will be left uncompleted, but is used by the attacker to complete the first one.Trudy is pretending to be Alice(messages are spoofed), so she sends to Bob “I’m Alice, this is my challenge R2”, Bob is correcting responding with encrypted R2, butTrudy cannot send back the encrypted R1 cause it doesn't know the key. So she starts another session with Bob with the challenge that received from Bob in the first session, and in the second session he will get the correct result of the encryption R1 with another challenge R3, that he will ignore. Bob can obviously suspect because the second session will be incomplete. We can have two different solutions,like using different keys for each session (KABandKAB+ 1) or different challenges.



Mutual Authentication with timestamps

It is crucial to alter the timestamp in response from Bob.
Also important is the **synchronization**.

Real world protocols: **ISO/IEC 9798-2**

Slightly variants of the previous protocols are defined by ISO/IEC 9798-2:

- one-pass unilateral authentication: $A \rightarrow B : E_K(ts_A, B)$ where ts_A is either a timestamp or a sequence number.
- two-pass unilateral authentication:
$$B \rightarrow A : N_B$$
$$A \rightarrow B : E_K(N_B, B)$$
where N_B is a nonce.
- two-pass mutual authentication:
$$A \rightarrow B : E_K(ts_A, B)$$
$$B \rightarrow A : E_K(ts_B, A)$$
This is just the composition of two unilateral authentications.
- three-pass mutual authentication:
$$B \rightarrow A : N_B$$
$$A \rightarrow B : E_K(N_A, N_B, B)$$
$$B \rightarrow A : E_K(N_B, N_A)$$

However, some **PRACTICAL PROBLEMS** income when a secret must be shared with each entity in the network: if users in the network are a lot, then the number of keys rises proportionally. For n users, we need $\frac{n \times (n-1)}{2}$ keys, and each user has to store $n - 1$ keys. When a new user joins the network, n keys must be securely transferred!

Some possible alternatives can be to share a secret with a **centralized trusted party** or to use **public-key** schemes.

Authentication based on a Trusted Party

Scenario:

- A and B share a secret key with C (K_{AC} or K_{Alice} and K_{BC} or K_{Bob})
- C is also called authentication server or Key Distribution Center (KDC)
- A and B might not be human user but also entity of the system (e.g., printer, databases etc.)

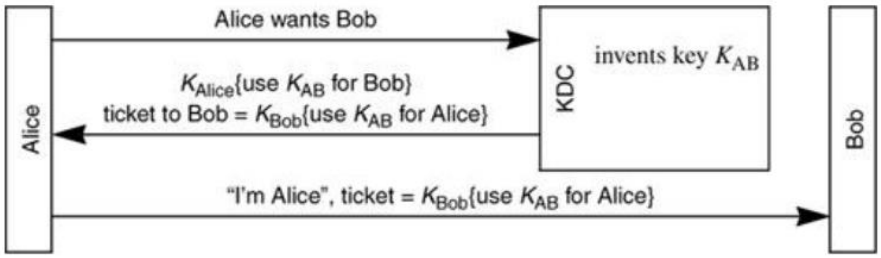
Goals:

- authenticate A (or A and B)
- [optional] decide on a session key K_{AB} to be used between A and B for short time, where K_{AB} is known only to A and B, randomly chosen and never used in the past communications

An attacker T **can...**	...but **can't**...
Be a legitimate user of the system, sharing a key with C. Sniff and spoof messages. Run more sessions with A, B, C concurrently. Convince A and/or B to start a new session with T. Might know old sessions.	Guess random numbers chosen by A or B Does not know keys K_{AC}, K_{BC} Is not able to decode in a short time messages encrypted with unknown keys.

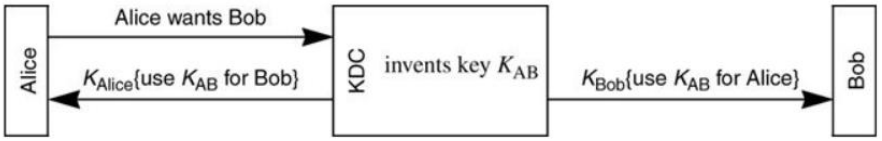
* **KDC** = Key Distribution Center

The **protocol** for authentication with a trusted party is:



A tells C that wants to talk with B.
C chooses the Key that A should use with Bob.
A talks to B.

Ticket: is the message generated by KDC for Bob, containing the session key, and it is sent to alice.
This protocol is weak **against MITM attacks**.



C needs to communicate with A, B.

Problem: this requires synchronization: A may send something encrypted to B before B has received the shared key.

MITM attacks on Authentication with a Trusted Party

1. A sends to T (instead of A sends to C) :
   A, B
2. immediately T sends to C:
   A,T
3. C chooses K and sends to A:
   $K_{AC}(K)$ and $K_{TC}(K)$
4. A sends to B:
   C, A, $K_{TC}(K)$
5. T intercepts the message sends to A:
   K(Hello A, this is B)

Problem: Alice believes to have a shared key with Bob, instead the key is with Trudy!

In this way Trudy is performing a Man In The Middle attack!

So the protocol can be **revised** to avoid this kind of attack:

1. A sends to C:
   $A, K_{AC}(B)$
2. C chooses K (secret shared key) and sends to A:
   $K_{AC}(K)$ and $K_{BC}(K)$
3. A decodes and computes K and sends to B:
   $C, A, K_{BC}(K)$
4. B decodes $K_{BC}(K)$ finds K and sends to A:
   K(Hello A, this is B)

Trudy now does not know that A wants to talk to B, since the request for talking with B is encrypted.

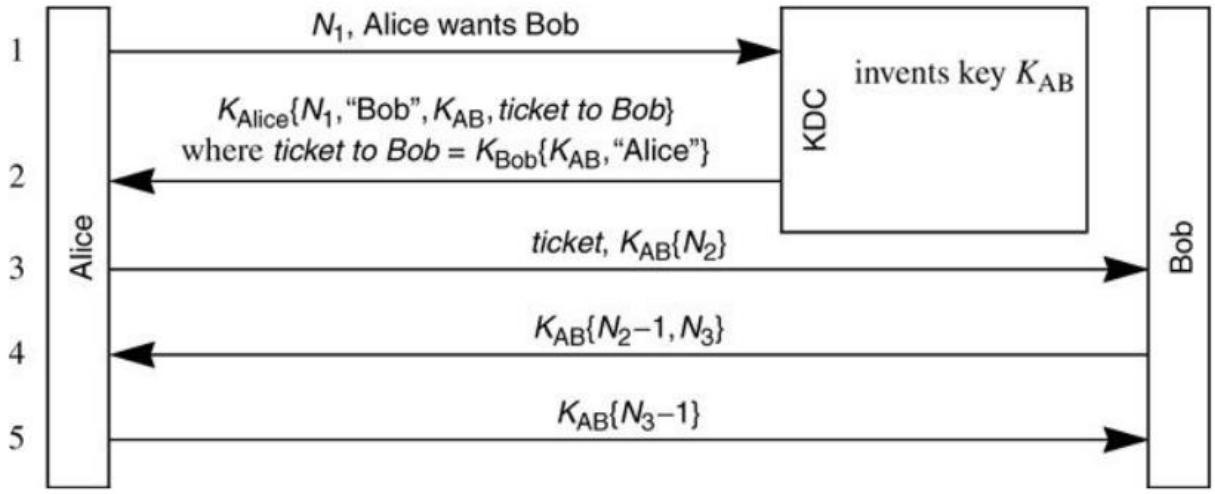
However this is not enough for Trudy to give up with his MITM attack.

A **Man In The Middle attack** can still be performed.

1. A sends to T (instead of A sends to C) :
   $A, K_{AC}(B)$
2. T intercept the message and replays old message:
   $A, K_{AC}(T)$
3. C chooses K and sends to A:
   $K_{AC}(K)$ and $K_{TC}(K)$
4. A sends to B:
   $C, A, K_{TC}(K)$
5. T intercepts the message, understand that A is trying to talk to B and sends to A:
   K(Hello A, this is B)

Problem: Alice believes to have a shared key with Bob, instead the key is with Trudy. The only difference now is that Trudy has to wait the end of the attack to know the identity B.

Needham-Schroeder (NS) Symmetric Protocol



NS REPLAY ATTACK (DENNING AND SACCO) :

1. A chooses N (nonce) and sends to C: A, B, N_1
2. C chooses K and sends to A: $K_{AC}(N_1, B, K_{AB}, K_{BC}(K_{AB}, A))$
3. A decodes, checks N_1 and B and sends to B: $K_{BC}(K_{AB}, A)$. $K_{AB}(N_2)$
4. T blocks the message for B, replays (as A) to B: $K_{BC}(K'_{AB}, A)$, $K'_{AB}(N''_2)$
5. B decodes, chooses nonce N_2 and sends to T (not to A): $K'_{AB}(N''_2\text{-}1, N_3)$
6. T sends to B: $K'_{AB}(N_3 - 1)$

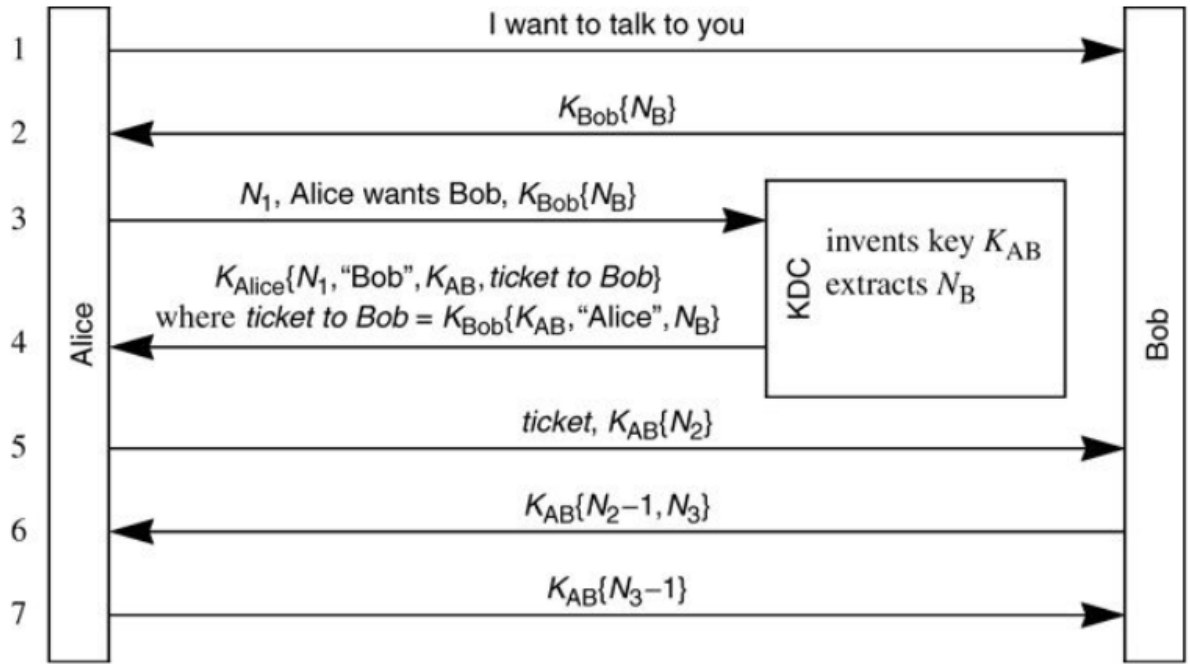
To carry a **replay attack** two sessions are needed:

- In **session #1** Trudy records $K_{BC}(K'_{AB}, A)$ and later he will be able to recover K'_{AB}.
- In **session #2** Trudy intercepts messages from A to B and reuses old $K_{BC}(K'_{AB}, A)$.

To **prevent replay attacks** timestamps can be used, making a message valid only for a small window of time. Messages can also be numbered so that A or B can recognize if T is using an older message. More, nonces can be mitigated.

Needham-Schroeder symmetric protocol can be fixed:

Mitigation based on a nonce.



Last message could be safely removed

1. A chooses N and sends to B:
   A, N
2. B chooses N' and sends to C:
   B, N', K_{BC}(N, A, t)
3. C sends to A:
   K_{AC}(B, N, K_{AB}, t) , ticket = K_{BC}(A, K_{AB}, t), N'
4. A sends to B:
   K_{BC}(A, K_{AB}, t), K_{AB}(N')

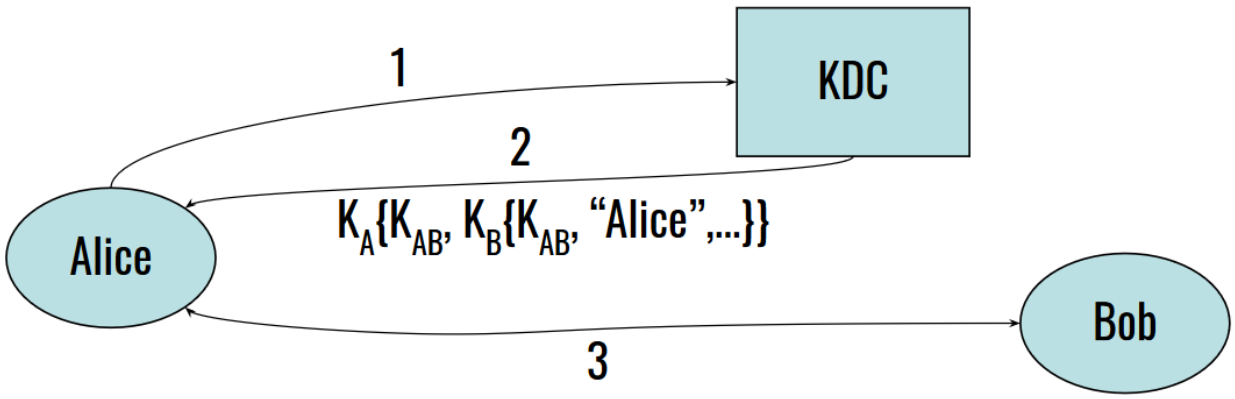
Kerberos v4/v5

Kerberos Is a general framework used in order to **provide authentication in distributed systems**, this framework is providing safe access to resources of a network.
The main idea of this framework is that users that want to use the service of another party have to make a **request to the KDC for a ticket**, and then they have to present this ticket to that party who offers the service. Of course **every user shares a secret key with KDC**.

Not only humans are users, many times also applications but the difference is that while humans have not strong capabilities and their keys are derived from passwords,applications can store long keys(not predictable) that are not stored in clear.

Each user (also named principal) has a **master secret** with KDC.
Each principal is registered by the KDC. All the master keys are stored in the KDC database and are encrypted with the KDC mater key.

A **ticket** is encrypted with the secret key associated with the service. A ticket contains some information such as sessionkey, username, client network address, servicename, lifetime, timestamp.



Ticket of Alice for Bob: $K_B\{K_{AB}, \text{“Alice”}, ...\}$,
K_A master key of Alice, K_B master key of Bob,
K_{AB} session key to be used by A and B
only Bob is able to decode and checks the message

A **simplified version** of Kerberos might be:

1. A sends to C : A, B, N
   So Alice asks to KDC “I am Alice, I want to talk with Bob, here is my nonce N”. The nonce will be used to verify the **authenticity** of C.

2. C responds to A : $[TicketB, K_A(K_{AB}, N, L, B)]$
   The KDC replies to Alice with a **ticket** encrypted with K_B (the master key of Bob) and

with the encryption by K_A (Alice's master key) of: the session key K_{AB}, the nonce N, the **lifetime** L and the identity of Bob.

3. A checks N and knows the ticket lifetime L and sends to B : $[TicketB, K_{AB}(A, t_A)]$
   Alice checks the nonce and learns about the ticket lifetime, then Alice sends to Bob the $TicketB$ and the **authenticator** which is $K_{AB}(A, t_A)$, where t_A is the **timestamp** of Alice.

4. B checks that A's identity in $TicketB$ and in the authenticator are the same and also for the time validity of the ticket.
   So Bob **decrypts** the ticket with his master key, obtaining the identity of Alice and the **session key**, then with the session key he will check if the **authenticator** and the **identity** of Alice corresponds to the identity of the ticket, and will also check if the ticket is still valid according to the **timestamp**.

5. B sends to A : $K_{AB}(t_A)$
   So Bob replies to Alice with the **encryption** of the **timestamp** with the **session key** K_{AB}.
   In this way Bob shows his knowledge of t_A.

However this protocol has **some problems**: messages between host and KDC should be protected using the master key (derived from the user's password).
Two solutions proposed for each request to KDC:

- User Alice must type the password each time
- User Alice's password is temporarily stored

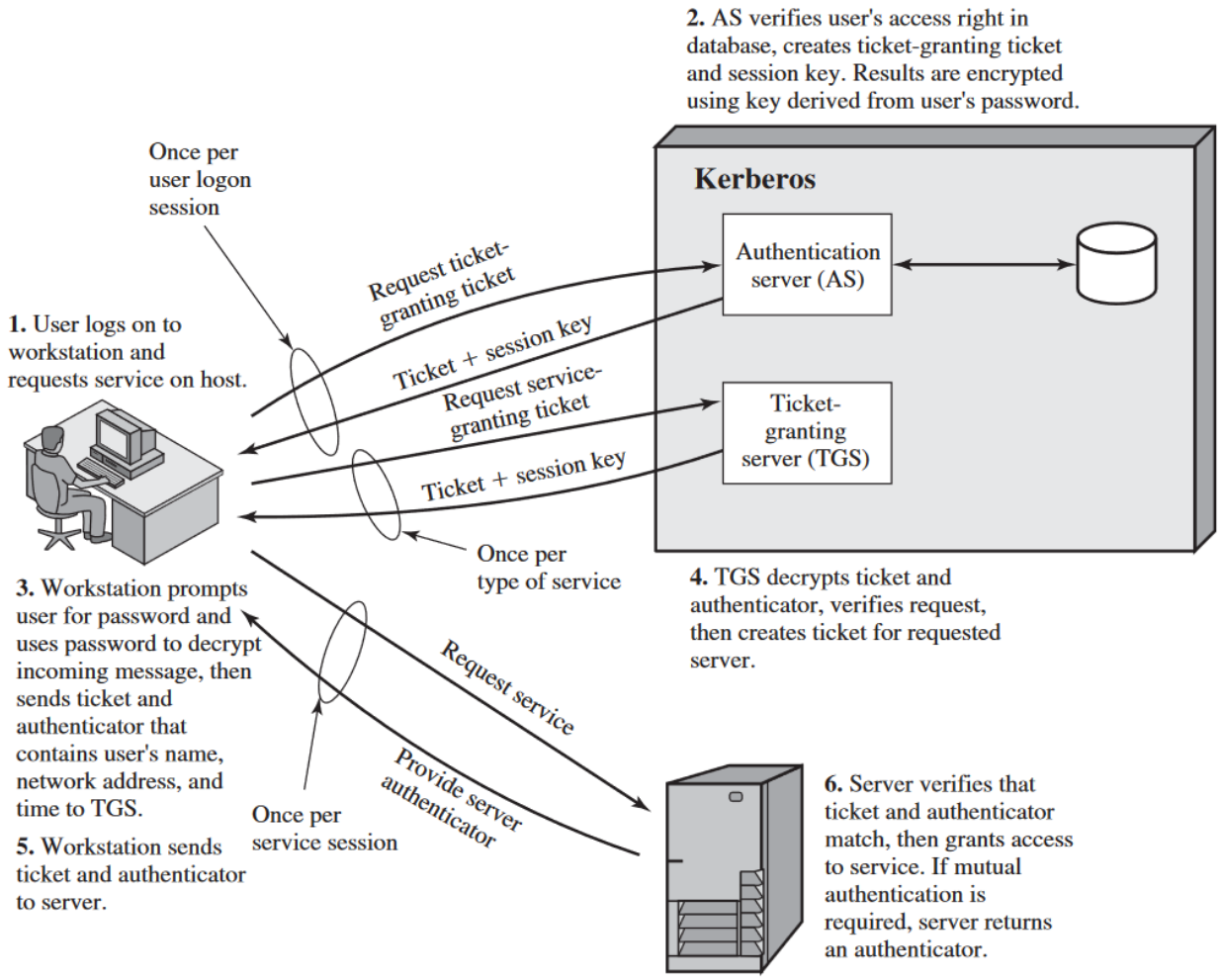
The two solutions are clearly **inadequate**.

Ticket-granting Ticket (TGT)

The idea is to use a meta-ticket to ask for a real ticket such that A is having a **ticket for asking tickets**, so that every time she needs a ticket she shows the meta-ticket instead of repeating the password.
It is useful to **reduce the number of times an user types the password**.

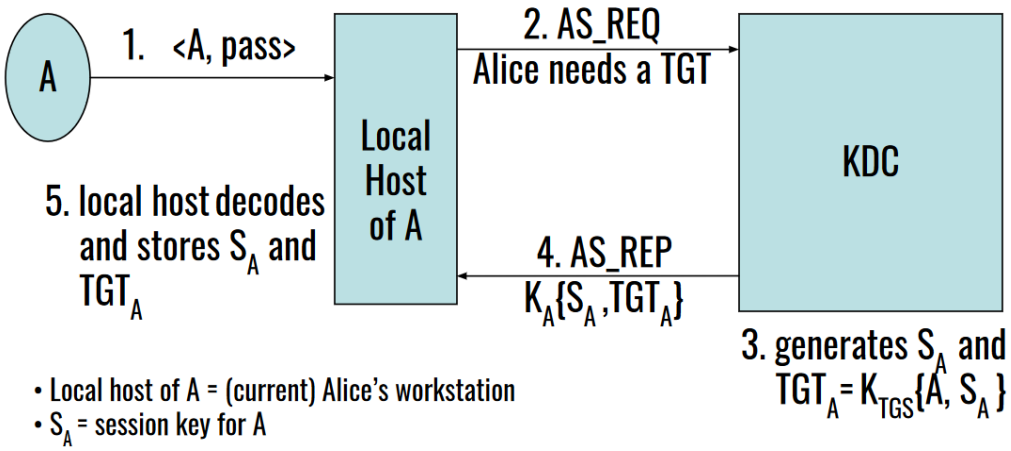
- In the initial login a session key S_A is derived for Alice by KDC. S_A has a fixed time (e.g. 1 day, 4 hours).
- KDC gives Alice a TGT that includes session key S_A and other useful information to identify Alice (encrypted with KDC's master key).
- Alice now wants to use a particular service, so che will request it by showing the TGT to the TGS (**ticket-granting server**, the server that provides tickets for services).
- TGS will answer with the requested ticket and a session key valid between Alice and the service.
- Alice now can request the service to the server cause he has the service-granting ticket.
- The server will answer with the server authenticator.

In this way, the password is **used only once** and it is **not stored**.

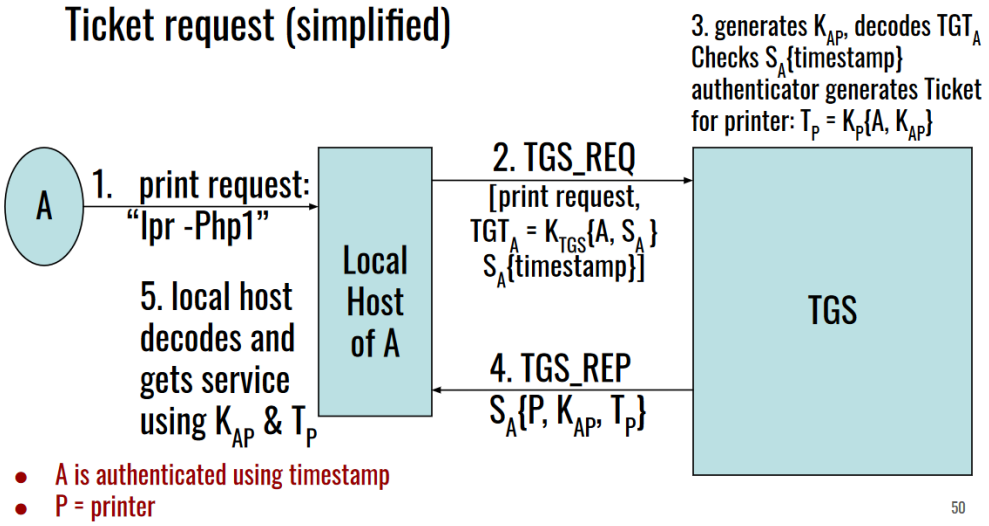


The whole operation can be summarized in **three phases**:

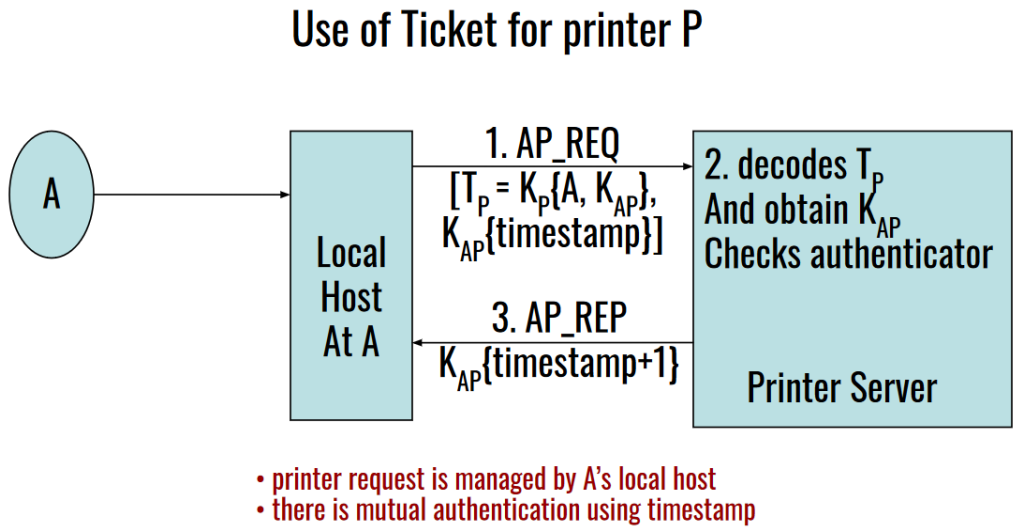
- **Login phase**, where Alice types the password, the KDC derives from the password a **session key** S_A and then KDC answers Alice with a TGT and S_A encrypted in K_A.
  So Alice will have a message like $K_A(S_A, TGT) = K_A(S_A, K_{KDC}("Alice", S_A))$



- **Ticket Request Phase**, where Alice requests a session key to KDC in order to talk with Bob, so she needs to be **authenticated** by sending the server the TGT and an authenticator. Once received, the server will **decrypt** TGT to get S_A and will also **decrypt** the authenticator in order to verify the timestamp.
  KDC will then find **Bob's master key** K_B and will create a session key K_{AB}, so with all these information it will create a ticket to Bob: $K_B("Alice", K_{AB})$.
  So KDC will send back to alice: $S_A("Bob", K_{AB}, ticket\ to\ Bob)$ encrypted in S_A.



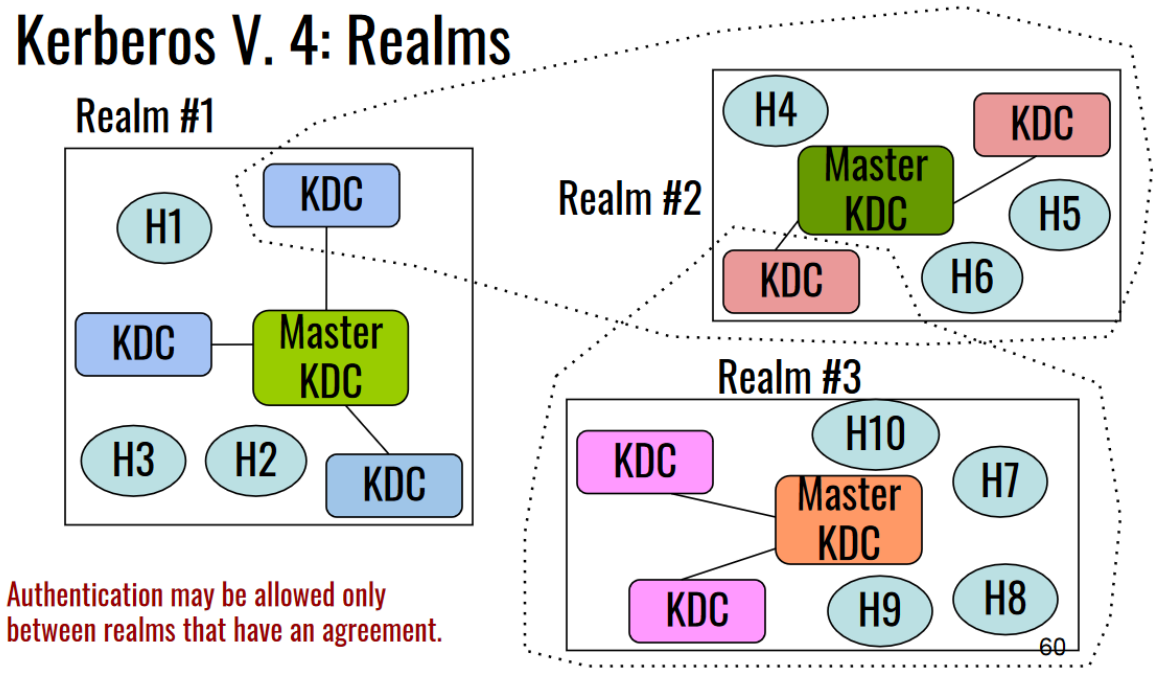
- **Ticket Usage Phase**, where Alice has the ticket and to use it she sends it to Bob, together with the timestamp encrypted by K_{AB} (like in the simplified version).
  So Bob will decrypt the ticket which contains the **session key** K_{AB} and so he will be able to decrypt the timestamp in order to **trust** Alice, and at the end will reply with an incremented timestamp to get authenticated by Alice.



KDC and TGS are similar. One KDC can serve different systems (1 KDC many TGS).

Kerberos Realms

In several cases, for security and performance reasons not only one domain is used, but several (i.e. several KDC) : here comes the **realms**.
Each realm has a different **master KDC** and all KDCs share the **same master key**.
Two different KDCs in different realms have different databases for users.



Examples and Exercises

EXERCISE 1 - Alice and Bob agreed to the special message T = "The cat is on the moon" and on a 256b secret key K. For one-way authentication Alice sends to Bob EncK(T): Bob decrypts, recognizes T and authenticates Alice. Discuss the security of the protocol and possible improvements .

Supposing that the key k is only known by A and B and no other, Alice can be truly authenticated by Bob in this way. However, an attacker can intercept the message and can send it again to bob whenever he wants, and Bob will think that Alice sent it, while it is wrong. So there is **no proof of freshness** in this protocol.
To cope with this problem, a **timestamp** or a **challenge mechanism** can be added to this protocol.

EXERCISE 2 - Alice and Bob discuss about the mutual authentication (not based on third parties). Bob claims that mutual authentication can be simply implemented by sequencing two (opposite) uni-directional authentications; Alice insists that it is better to implement a protocol authenticating both parts, because of possible attacks in the time interval between the two authentications. Describe and motivate your position.

There is no easy and unique answer as it depends on how we design the protocol. In general, we have to require freshness of the messages. However, this could not be enough: e.g., ISO/IEC 9798-2:

- uni-directional authentication: $A \rightarrow B : E_K(ts_A, B)$
- mutual authentication with two unilateral authentications:
$$A \rightarrow B : E_K(ts_A, B)$$
$$B \rightarrow A : E_K(ts_B, A)$$

We can see that **both protocols ensure freshness** of the messages **by using a timestamp**, hence in theory they prevent replay attacks.
However, the second protocol can be attacked by C (where C(X) means that C is impersonating X):

$$(1)\ A \rightarrow B : E_K(ts_A, B)$$
$$(2)\ B \rightarrow C(A) : E_K(ts_B, A)$$
$$(1')\ C(B) \rightarrow A : E_K(ts_B, A)$$
$$(2')\ A \rightarrow C(B) : E_K(ts'_A, B)$$

A is involved into two sessions:
- in the first one as the initiator
- in the second one as the responder

The attacker is exploiting the confusion in the protocol between the two roles and **reuse messages from one session into the other**. To fix this problem, the protocol has to provide "**injective agreement**", for instance by adding in some way the role of the creator of a message into the message. During the course, we said that for instance the initiator can use K and the responder K+1.

Using a (true) mutual authentication protocol that “binds” the initiator challenge with the responder challenge can mitigate attacks based on role mixup. For instance, in ISO/IEC 9798-2:

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

EXERCISE 3 - Kerberos

1. Discuss what a "ticket" is and what benefits it provides.
2. Discuss what a "ticket-granting ticket" is and what benefits it provides.
3. What is a realm? Describe when and how it is possible to get services from some principal belonging to another realm.

1. When user A wants to use a service in the Kerberos network she asks the KDC for a ticket and then she presents that ticket to the party who offers the service.
   When A asks the KDC a ticket to communicate with B, the KDC replies to A with a ticket encrypted with $K_B$that A will send to B with some other information.
   By decrypting the ticket with his key, B assures that A followed the protocol contacting the KDC and she has been authorized.
2. The idea of the TGT is to use a meta-ticket to ask for a real ticket. This meta-ticket is given to A and has a specific validity for that session (maybe it lasts some hours). Alice can show this meta-ticket to the Ticket Granting Server (TGS) and receives the ticket to use the service she asked for.
   Why to use a TGT? When Alice asks the KDC for a ticket she has to authenticate with it, maybe with a password. With the TGT Alice obtained this limited-lifetime ticket so that she doesn’t need to type her password for each authentication!
3. A realm is a distinct Kerberos network with a KDC/TGS. Various realms can be organized with different domains and services, but also in such a way that an user can login with his KDC without sending his password to another KDC.

Authentication III - protocols based on public keys

Using standard **public key scheme** to sign messages containing challenges/timestamps i.e. :

- A to B : $A, B, T\ (timestamp), N\ (nonce)$
- B to A : $K_{BPrivate}(A, B, t, N + 1)$
- A checks (A, B, t, N) in the message using $K_{BPublic}$

Can be not so easy: if Trudy can convince Alice that public key of Bob is $K_{TPublic}$ (the public key of Trudy, actually), then the system does not work.

Solution: Public Key Infrastructure (**PKI**), i.e. a system that integrates an authority able to guarantee correctness of public keys by signing them.

Nedham-Schroeder public-key protocol

Where C is the **trusted authority** that signs public keys.

Mutual Authentication:

1. A to C: A, B
2. C to A: B, K_{PB_B}, $Sig_C(K_{PB\_B}, B)$
3. A checks digital signature of C, generates nonce N and sends to B: $K_{PB\_B}(N, A)$
4. B decodes it and (now wants to check A's identity) sends to C: B, A
5. C to B: A, K_{PB_A}, $Sig_C(K_{PB\_A}, A)$
6. B checks C's digital signature, retrieves K_{PB_A}, generates nonce N' and sends to A: $K_{PB\_A}(N, N')$
7. A decodes, checks N, and sends to B: $K_{PB\_B}(N')$

However a **MITM attack** can be performed: if Trudy is a system user that can talk (being authenticated) to A, B and C, then T can (**S1**) authenticate with A and then(**S2**) can authenticate with B faking to be A:

- A to T [step 3 of **S1**]: $K_{TPub}(N, A)$
- T(like A) to B [step 3 of **S2**]: $K_{BPub}(N, A)$
- B to T(like A) [step 6 of **S2**]: $K_{APub}(N, N')$
- T to A [step 6 of **S1**]: $K_{TPub}(N')$
- A to T [step 7 of **S1**]: $K_{TPub}(N')$
- T(like A) to B [step 7 of **S2**]: $K_{BPub}(N')$

Thus, it would be useful to **fix** the protocol (see **point 6**):

Mutual authentication:

1. A to C: A, B
2. C to A: B, K_{PB_B}, $Sig_C(K_{PB\_B}, B)$
3. A checks digital signature of C, generates nonce N and sends to B: $K_{PB\_B}(N, A)$
4. B decode (now wants to check A's identity) and sends to C: B, A
5. C to B: A, K_{PB_A}, $Sig_C(K_{PB\_A}, B)$
6. B checks C's digital signature, retrieves K_{PB_A}, generates nonce N' and sends to A: $K_{PB\_A}(B, N, N')$
7. A decodes, checks N, and sends to B: $K_{PB\_B}(N')$

In this way, the previous MITM attack **does not work**:

- A to T [step 3 of **S1**]: $K_{TPub}(N, A)$

- T(like A) to B [step 3 of **S2**]: $K_{BPub}(N, A)$
- B to T(like A) [step 6 of **S2**]:
  BEFORE : $K_{APub}(N, N')$
  NOW : $K_{APub}(B, N, N')$

- T to A [step 6 of **S1**]:
  BEFORE : $K_{TPub}(N')$
  NOW : **T cannot send** $K_{APub}(B, N', N)$ **while talking to A!**
- A to T [step 7 of **S1**]: $K_{TPub}(N')$

- T(like A) to B [step 7 of **S2**]: $K_{BPub}(N')$

X.509 Authentication Standard (RFC 5280)

Defined in 1988 and several times revised (until 2000).
We need a **directory** of public keys by certification authority.
This standard defines protocols for authentication, **not encryption**.
Three authentication protocols are defined in this standard: **One-Way, Two-Ways, Three-Ways**.

One-Way authentication

A to B: $cert_A$, D_A, $Sig_A(D_A)$

where:

- $cert_A$ is the certificate of A's public key, signed by an authority
- D_A is (t, B, $K_{PB\_B}(K_{AB})$)
- t is a timestamp
- K_{AB} is a shared secret key
- $Sig_A(D_A)$ is a signuture on D_A from A

If Alice wants to be authenticated by Bob she sends a single message containing the **digital certificate** of Alice obtained by the trusted authority, the **digital signature** of D_A and $D_A$that contains the timestamp, the identity of B, the session key K_{AB} encrypted with the public key K_{BPub} of B.
So a single transfer of information from A to B establishes the **identity** of A and the fact that the message **was generated by** A, the fact that the message is **intended for** B, and the **integrity and originality** (not sent multiple times) of the message.

Two-Ways authentication

1. A to B: $cert_A$, D_A, $Sig_A(D_A)$
2. B to A: $cert_B$, D_B, $Sig_B(D_B)$

where:

- $cert_X$ is the certificate of X's public key, signed by an authority
- D_A is (t_A, N, B, $K_{PB\_B}(K_{AB})$), D_B is (t_B, N, N', B, $K_{PB\_A}(K_{BA})$)
- t_X is a timestamp from X, N/N' are nonces to prevent replay attacks
- K_{AB} and K_{BA} are shared secret keys
- $Sig_X(D_X)$ is a signature on D_X from X

Remark. There is no identity of A in D_A: source of problems in NS public-key protocol.

There will be a message from Alice to Bob and another message from Bob to Alice.

Three-Ways authentication

Used in case of unsynchronized clocks (t=0 in the protocol):

1. A to B: $cert_A$, D_A, $Sig_A(D_A)$
2. B to A: $cert_B$, D_B, $Sig_B(D_B)$
3. A to B: B, $Sig_A(N, N', B)$

where:

- $cert_X$ is the certificate of X's public key, signed by an authority
- D_A is (0, N, B, $K_{PB\_B}(K_{AB})$), D_B is (0, N, N', B, $K_{PB\_A}(K_{BA})$)
- N/N' are nonces to prevent replay attacks
- K_{AB} and K_{BA} are shared secret keys
- $Sig_X(D_X)$ is a signature on D_X from X

This is the **most robust version**: authentication is based on **nonces**, and it is useful for **non-synchronized clocks**.
Three different messages are exchanged. In the last message A sends the two nonces and the identity of Bob signed by her digital signature.

REMARK: In X509, certificates are signed by a **Certificate Authority (CA)**. The infrastructure to support interactions with a CA and user authentication is defined as **Public Key Infrastructure (PKI)**.

Public Key Infrastructure (PKI)

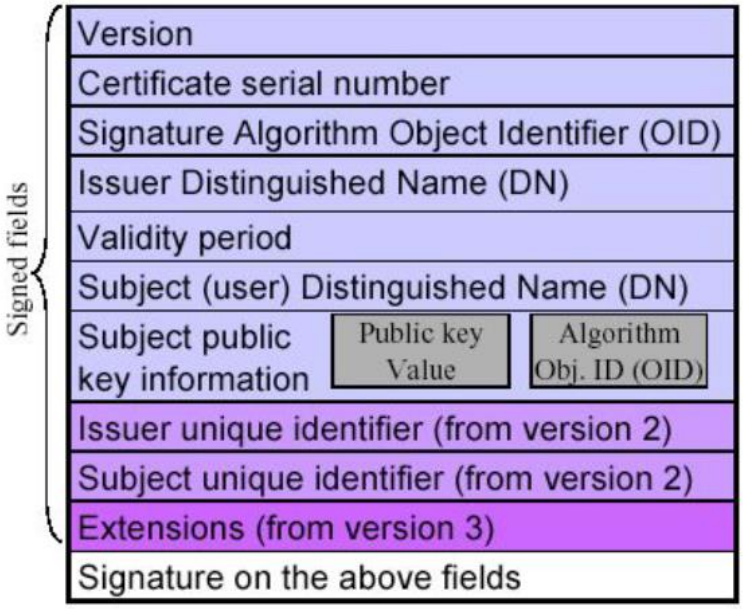
Certificates are issued by a trusted Certification Authority (CA), that provides certificates of all users in its domain. When someone wants to know the public key of a user, it asks the CA.
CA provides the user's public key, signing it by its own private key.
Keys are not used forever, they are subject to changes.

Three main kinds of certificate exist:

- **Extended Validation (EV)**: expensive, take several days (e.g. for Governments)
- **Organization Validation (OV)**: a few checks, reasonably fast (e.g. for big organizations)
- **Domain Validation (DV)**: the applicant has the right to "use" a specific domain name. Very common and fast. (e.g. for websites)

CA run by Internet Security Research Group (ISRG) that provides X.509 certificates for Transport Layer Security (TLS) encryption at no charge. **Let's Encrypt** certificates are valid for 90 days, during which renewal can take place at any time.

The X.509 certificate has **many fields**:



- VERSION. There are currently three versions defined, version 1 for which the code is 0, version 2 for which the code is 1, and version 3 for which the code is 2.

- SERIAL NUMBER. An integer that, together with the issuing CA's name, uniquely identifies this certificate.

- SIGNATURE ALGORITHM ID. Specifies the algorithm used to compute the signature on this certificate. It consists of a subfield identifying the algorithm followed by optional parameters for the algorithm.
- ISSUER NAME. The X.500 name of the issuing CA. It follow a specific format and provides several details:
  - C: country, e.g., US
  - OU: organization unit, e.g., company name
  - CN: common name, e.g., *.google.com

  There is no standard on how to display these kinds of information.

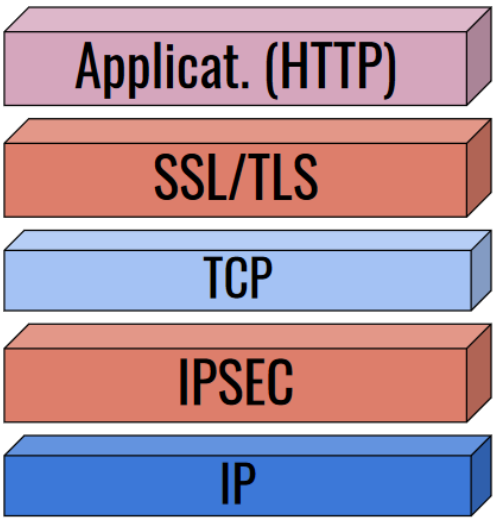
- VALIDITY. This contains two subfields, the time the certificate becomes valid, and the last time for which it is valid.

- SUBJECT. The X.500 name of the entity whose key is being certified. Again it follows specific rules.
- SUBJECT PUBLIC KEY INFO. This contains two subfields: (a) an algorithm identifier, and (b) the subject's public key.
- ISSUER UNIQUE IDENTIFIER. Optional (permitted only in version 2 and version 3, but deprecated). Uniquely identifies the issuer of this certificate.
- SUBJECT UNIQUE IDENTIFIER. Optional (permitted only in version 2 and version 3, but deprecated). Uniquely identifies the subject of this certificate.
- EXTENSIONS. These are only in X.509 version 3. X.509 allows arbitrary extensions, since they are defined by OID.
- ENCRYPTED. This field contains the signature on all but the last of the above fields.

IPSec

KEY POINTS

- IP security (IPsec) is a capability that can be added to either current version of the Internet Protocol (IPv4 or IPv6) by means of additional headers.
- IPsec encompasses three functional areas: authentication, confidentiality, and key management.
- Authentication makes use of the HMAC message authentication code. Authentication can be applied to the entire original IP packet (tunnel mode) or to all of the packet except for the IP header (transport mode).
- Confidentiality is provided by an encryption format known as encapsulating security payload. Both tunnel and transport modes can be accommodated.
- IKE defines a number of techniques for key management.



Approaches:

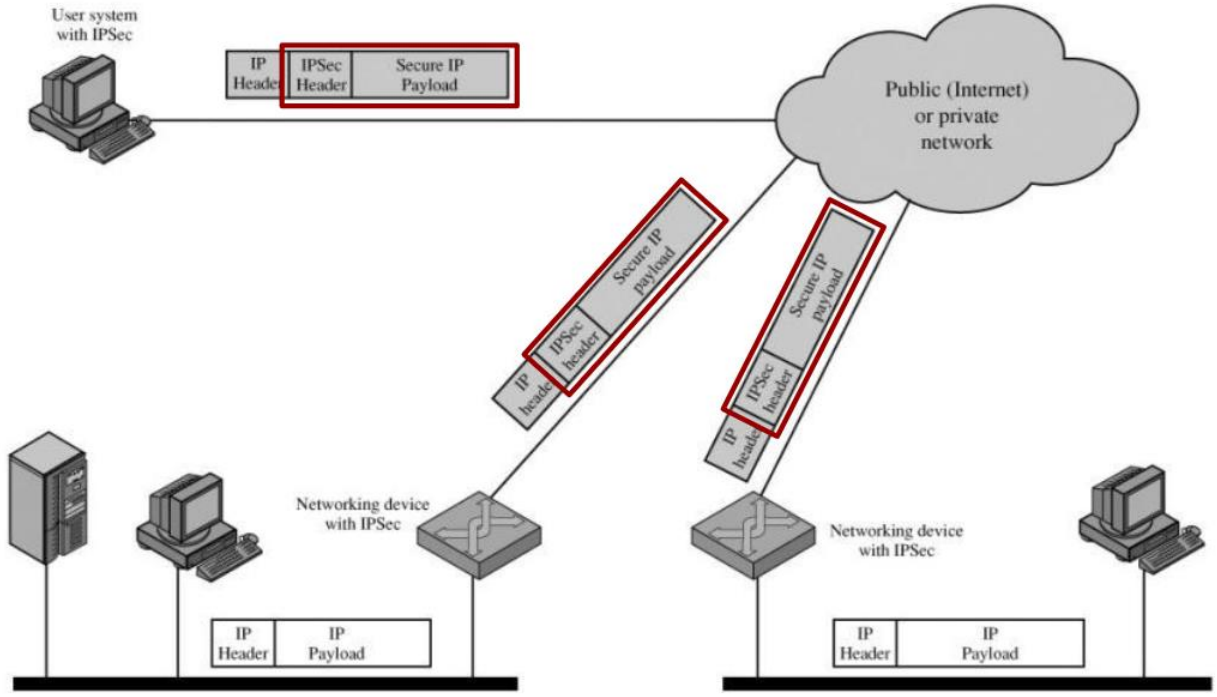
- Security in the applications: PGP, HTTPS, S-HTTP, SFTP.
- Security in the protocol stack:
  - SSL/TLS between TCP and application layer
  - IPSEC between IP and TCP

In the **protocol stack** we have many levels: the **application layer,** the **transport layer TCP**, then the **network layer IP**.
IPSEC works **between TCP and IP** layers, so we say that TCP uses IPSEC.
IPSEC protects IP packets at each hop, in fact there is a shared key among two routers that are connected by a link, so all traffic is encrypted (even IP headers), but this requires cooperation among routers and a significant computational effort.
IPSEC is very important because it is a security protocol that can be used by all applications.

IPSEC is used for:

- **Authentication** : So we are able to understand who are the two parties that are exchanging messages, and also implies data integrity in some sense.
- **Confidentiality** : Refers to protecting information from being accessed by unauthorized parties, so all the people who are authorized to do so can gain access to sensitive data.
- **Key Management** : A standard for generating session keys used during the communication, and other keys. (not studied here)

IPSEC is applicable over LANs, across public and private WANs and for the internet.
It is mandatory for IPv6 (since 2011) and optional in IPv4.
IPSEC is also used in VPNs. Other protocols for VPNs are *Wireguard* or *OpenVPN*.



In the figure it is possible to appreciate some uses of IPSEC between different LANs.
Within a LAN, packets are seen just like normal packets, but between different LANs what changes is the IPSEC header that can help to distinguish whether someone is in one LAN or in another, but what is in the two LANs stays unknown.

With IPSEC some choices are possible:

- You can set up your gateway (firewall/router) in order to provide strong security for the traffic out-coming and incoming your gateway, so you do not need to protect the whole local traffic, it is sufficient to **protect the gateway**.
- You cannot bypass the gateway just because it is not working, **the attacker is not able to change the behavior of the gateway** from a remote position.
- it is **transparent** to applications, so changing things at low levels okay because applications will continue to work in the same way.

- It is **transparent** to end users, so it permits a VPN,virtual private network.
- The principal feature is that it can **encrypt and/or authenticate all traffic at the IP level**.

This is possible due to the **services** provided by IPSEC:

- Access control
  - prevents unauthorized use of a resource (computing cycles, data, network, bandwidth etc.)
- Connectionless integrity
  - detects modification of an individual IP datagram
- Data origin authentication
  - verifies the identity of the claimed source of data
- Rejection of replayed packets
  - a form of partial sequence integrity
- Confidentiality (encryption)
- Limited traffic flow confidentiality
  - *traffic flow confidentiality* = concealing source and destination addresses, message length, or frequency of communication

IPSEC is implemented as an **extension header** that follows the **main IP header**.This header is composed by two parts:

- **AH (Authentication Header)** : is the extension header for authentication and integrity. It is based on HMAC and requires a shared secret key.
- **ESP (Encapsulating Security Payload)** : that is the header used for encryption. However besides encryption, with ESP we can also get authentication.

	AH	ESP (encryption only)	ESP (encryption plus authentication)
Access control	✔	✔	✔
Connectionless integrity	✔		✔
Data origin authentication	✔		✔
Rejection of replayed packets	✔	✔	✔
Confidentiality		✔	✔
Limited traffic flow confidentiality		✔	✔

Remark. ESP does not allow inspection of inner data (which can useful in, e.g., firewalls)

IPSEC is not a standard, but a **family of standards**.

Based on three main concepts:

- Security Association (SA): provides the bundle of algorithms and data that provide the parameters necessary for IPSec operations. Internet Security Association and Key Management Protocol (ISAKMP) has been defined to establish SA.
- Authentication Headers provides connectionless data integrity and data origin authentication for IP datagrams and provides protection against replay attacks.
- Encapsulating Security Payload (ESP) provides confidentiality, connectionless data integrity, data-origin authentication, an anti-replay service (a form of partial sequence integrity), and limited traffic-flow confidentiality.

Security Associations (SA)

SA is a **one-way association** between sender and receiver that affords security for traffic flows.
Defines the parameters of sending datagrams from Alice to Bob.
All security associations are stored into a database called **SADB**.
A SA is defined by some main parameters stored in the SADB:

- **SPI (Security Parameter Index)** : it is a pointer to information about the security you are going to use. It is a unique identifier, conceptually similar to TCP port number; it **enables the receiving system** to select the SA under which a received packet will be processed.
- **IP Destination Address**
- **Security Protocol Identifier :** tells whether you are using **ESP** or **AH** (if you want both, you will need two SA).
- **Sequence Number Counter** : 32-bit value used to generate the Sequence Number field in AH or ESP headers.
- **Sequence Counter Overflow**: flag indicating whether overflow of the Sequence Number Counter should generate an auditable event and prevent further transmission of packets on this SA.
- **Anti-Replay Window:** used to determine whether an inbound AH or ESP packet is a replay.
- **AH Information:** authentication algorithm, keys, key lifetimes, and related parameters being used with AH.
- **ESP Information:** encryption and authentication algorithm, keys, initialization values, key lifetimes, and related parameters being used with ESP.
- **Lifetime**: a time interval or byte count after which an SA must be replaced with a new SA (and new SPI) or terminated, plus an indication of which of these actions should occur.
- **IPSEC Protocol Mode: tunnel, transport** or **wildcard.**

- **Path MTU:** any observed path maximum transmission unit (maximum size of a packet that can be transmitted without fragmentation) and aging variables.

SADBs can suffer of **Fragmentation Attack**, a typical DoS attack, in which I'm sending data fragmented in datagrams, you know that fragments have header containing bits that say "this is lastfrag" and so on such that the receiver can reconstruct the order of packets. The concerned attack is based on faking the sequence number of fragments, so the attacker changes the offset or the bit that says whether the fragments are the last or not.

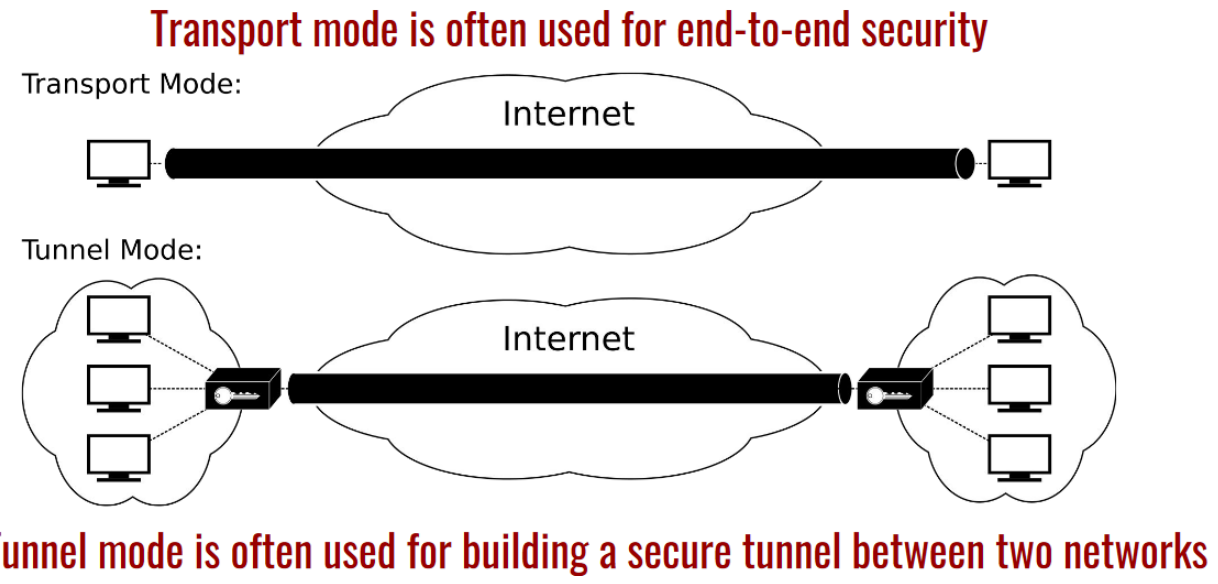
More than Security Association Databases (SADB) we also may have **Security Policy Databases (SPDB).** The SPDB contains a set of rules that determines whether a packet is subject to IPsec processing and governs the processing details. Each entry in the SPD represents a policy that defines how the set of traffic covered under the policy will be processed. Any inbound or outbound packet is processed in one of three ways: **discard, perform IPSEC processing, bypass IPSEC processing**.

Where are these DB? They are on both sides where we establish connections: on side A and on side B. They store the policies we need for the connections.

To match a packet with SA, SPDB uses **SA Selectors**: given a packet, through selectors we can identify it and establish what to do with it and how to handle it.

The **difference between SADB and SPDB** is that
SADB is a security association table, containing parameters of the security associations, while **SPDB** specifies the policies that determine the disposition of all IP traffic inbound or outbound from a host or a security gateway.

Transport vs Tunnel Mode

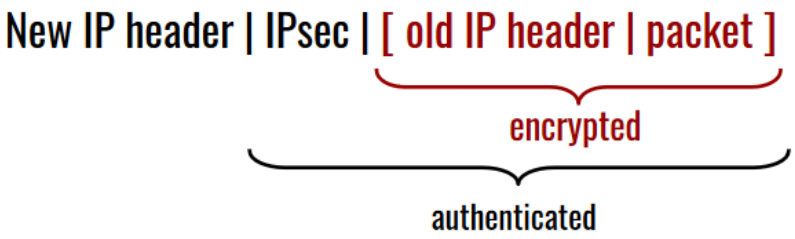


Transport mode is between **two machines** while tunnel mode is between **two networks**.

In **transport mode** we reuse the original IP header, while in **tunnel mode** a new IP header is used.

In **transport mode** only the payload is encrypted or authenticated, not the header: everybody can see the sender and the receiver: **routing is intact**, since the IP header is neither modified or encrypted.
When you perform authentication you check fields of the original IP header. This is good because you can know who the sender is, but some protocols may not work, such as in NATs, so you need to have a more complex configuration.
So this mode is used for **host-to-host** communication.
Transport and application layers are always secured by hash, so they cannot be modified in any way (for example by translating the port numbers).

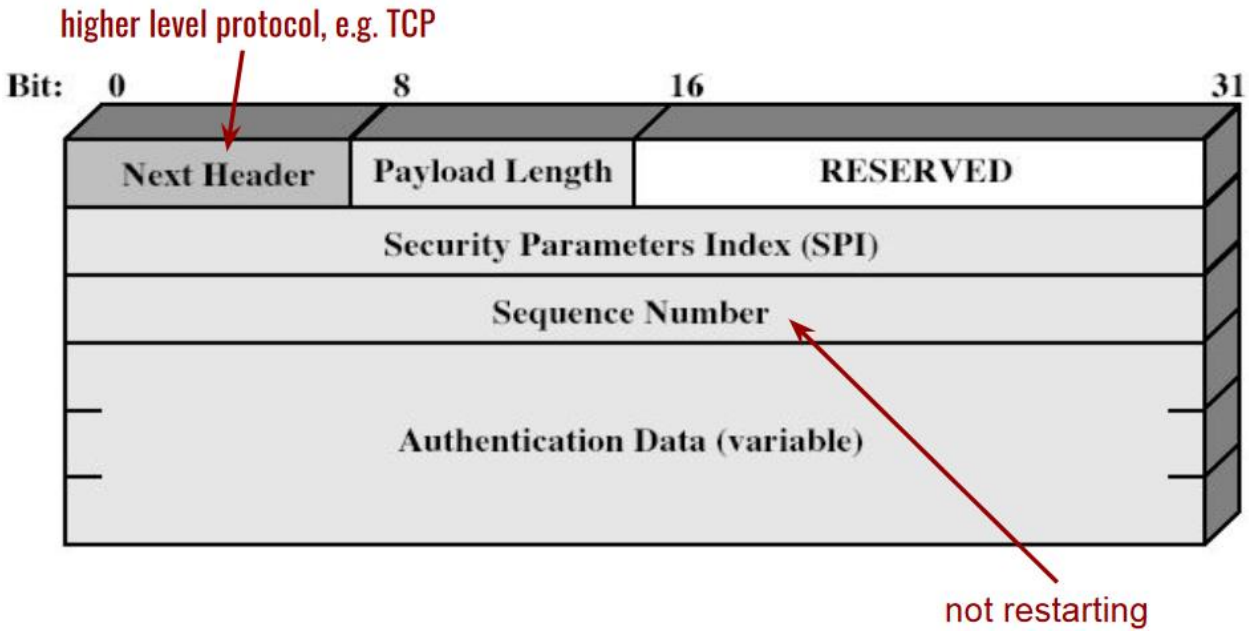
In **tunnel mode** you build a new IP header and you keep intact the original packet.
So you encrypt the IPSEC and the original packet. We expect that the router for network B can decrypt what network a sent.
This mode can be used when IPSec is applied at an intermediate point along path (e.g., for firewall-to-firewall traffic) and treats the link as a secure tunnel. Results in a slightly longer packet.
The entire IP packet (data and IP header) **is encrypted** and/or authenticated. It is then encapsulated into a new IP packet with a new IP header.
This is a **VPN**, for network-to-network communication.



	Transport Mode SA	Tunnel Mode SA
AH	Authenticates IP payload and selected portions of IP header and IPv6 extension headers.	Authenticates entire inner IP packet (inner header plus IP payload) plus selected portions of outer IP header and outer IPv6 extension headers.
ESP	Encrypts IP payload and any IPv6 extension headers following the ESP header.	Encrypts entire inner IP packet.
ESP with Authentication	Encrypts IP payload and any IPv6 extension headers following the ESP header. Authenticates IP payload but not IP header.	Encrypts entire inner IP packet. Authenticates inner IP packet.

Authentication Header (AH)

AH provides support for **data integrity** and **authentication** IP packets, but it **doesn't provide confidentiality** (i.e. does not encrypt data).
End system/router can authenticate users/apps. Also prevents address spoofing attacks by tracking sequence numbers.

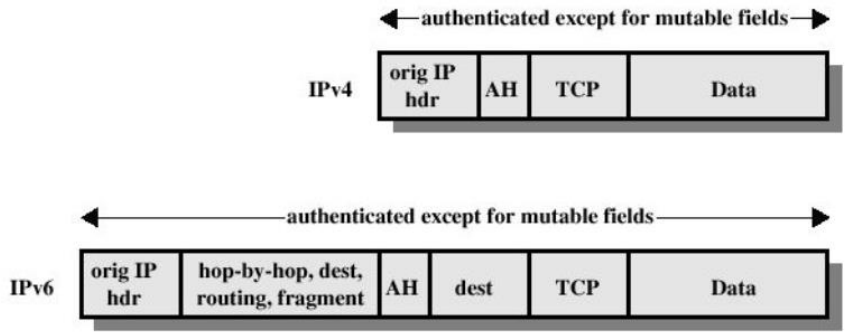


AH will perform the checksum of the IP payload, except for mutable fields of the IP protocol (e.g. time-to-live).

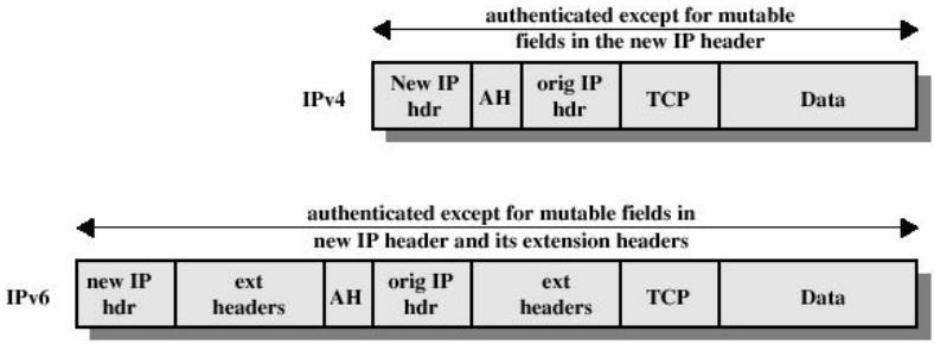
- **Next Header** identifies the higher level protocol that is using this protocol.
- **SPI** (Security Parameters Index) is something describing some technical details about the keys and other techniques used to implement the cryptography primitives needed to ensure security, but the way those bits are used is not officially set, so you can set up your scheme.
- **Sequence Number** is self explaining.
- **Authentication Data** explains the information about authentication, for example which hashing function is used and so on.

This Protocol Is protecting the payload, it **applies the MAC** function on the whole payload, but also some part of the IP header. In particular **only the non-mutable fields** (there are mutable and non-mutable fields, for example fields like the one that informs about fragmentation can vary packet to packet) **are authenticated and secured**. In this way AH can provide **data integrity** and **data origin authentication** (the IP address cannot be modified as this always invalidates the hash value).
However, it is incompatible with NAT, because once the packet reaches the gateway, the gateway cannot change IP destination in order to exchange it with the private one within its network, because this will invalidate the authentication.

In AH in **transport mode** the outcome is a packet that has an original IP header, the AH header and finally the payload (which is composed by TCP header and the Data)**:**

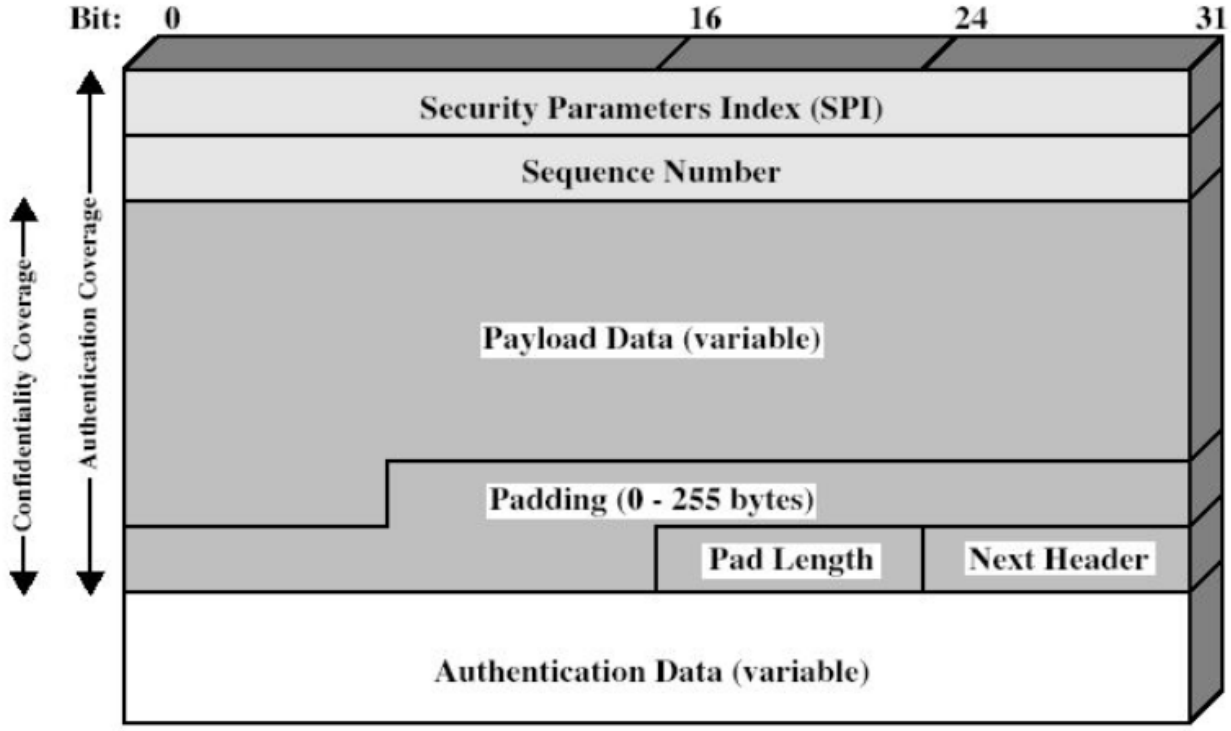


In AH in **tunnel mode** the original datagram (original IP header,TCP header and Data) becomes the new payload, then there is theNew IP header, and between them there is theAH header:

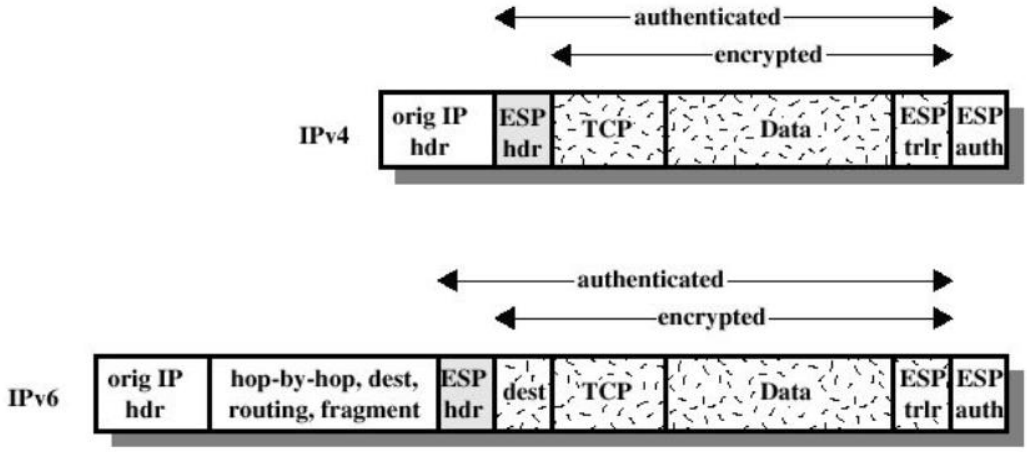


Encapsulating Security Payload (ESP)

ESP provides **encryption**, supporting all modern encryption algorithms like AES, Triple-DES and so on, and often CBC is used. The difference from AH is that when using the authentication feature of ESP, the authentication/data integrity is made **only on the payload**, not even the non-mutable fields of the IP header are included. This allows the ESP to be compatible with **NAT**. In some cases ESP provides also limited traffic flow confidentiality, traffic flow is something an attacker is interested to understand, he wants to see the type of traffic outgoing to your computer, you want to avoid this so you need **traffic flow confidentiality**.

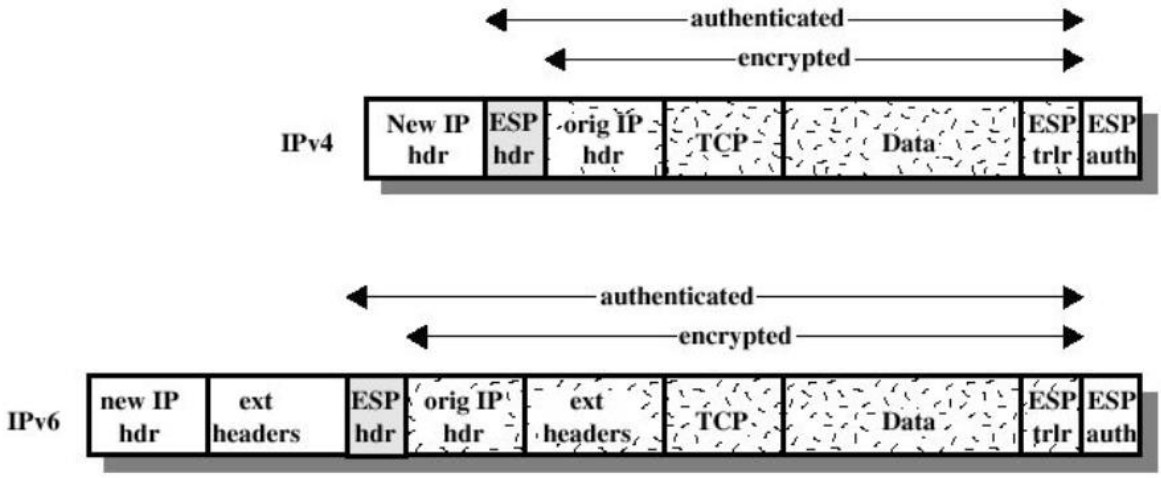


When ESP is used in**Transport Mode**, then the whole payload and the trailer is encrypted, and we have the original IP header, the ESP header, the payload and the ESP authentication portion in case of ESP. The ESP wraps the payload between header and a trailer:



(a) Transport Mode

When ESP is used in **Tunnel Mode**, the whole original datagram is encrypted, and so the final structure will be composed by New IP Header, ESP Header,payload and ESP auth portion in case of ESP:



TRANSPORT vs TUNNEL MODE ESP
Transport mode is used to encrypt and optionally authenticate IP data. Data is protected but the header is left in clear. This configuration can be good for host to host traffic. Adversary can try traffic analysis.
Tunnel mode encrypts the entire IP packet. It adds a new header for the next hop. It is slow. Good for VPNs.

Combining Security Associations

Because an ESP header cannot authenticate the outer IP header, it is useful to **combine an AH and an ESP**. SAs can implement either AH or ESP. This combination is called **security bundle.** There are two types of security bundle:

- **Transport Adjacency:** applying more than one protocol to the same IP packet, without tunneling. There's only one level of combination, further nesting yields no added benefit.
- **Iterated Tunneling:** application of multiple layers of security effected through multiple IP tunneling. This allows for multiple levels of nesting, since each tunnel can originate or terminate at a different IPSEC site along the path.

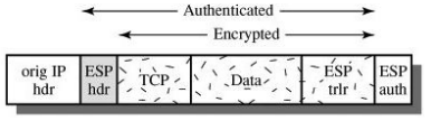
The two approaches **can be combined**, for example, by having a transport SA between hosts travel part of the way through a tunnel SA between security gateways.
Encryption and **authentication** can be combined in order to transmit a packet that has both **confidentiality** and **authentication** with several approaches: **ESP with authentication (ESPa)**, **transport adjacency** and **transport-tunnel bundle**.

ESP with Authentication option

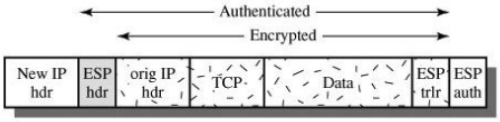
The simplest case is **ESPa**, where we apply ESP to data to be protected and after we append the authentication data field, and we can have two sub cases: **transport mode ESP**, where

auth and encryption are applied to IP payload, but IP header is not protected, and **tunnel mode ESP**, where auth and encryption are applied to the entire IP packet.

- User first applies ESP to data to be protected, then appends the authentication data field. Two subcases:
  - Transport mode ESP
    Authentication and encryption apply to IP payload delivered to the host, but IP header not protected



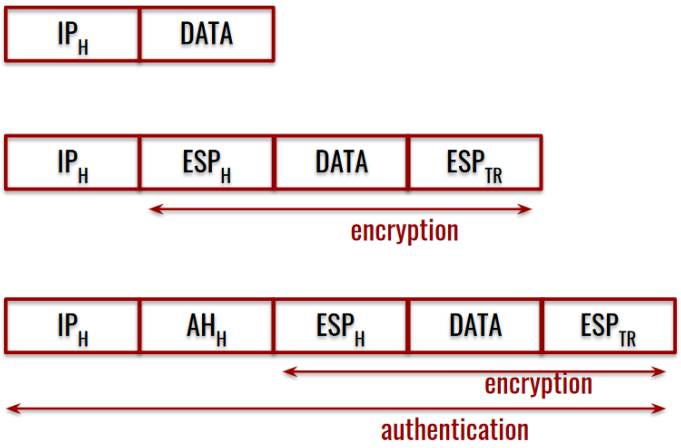
  - Tunnel mode ESP Authentication applies to entire IP packet delivered to outer IP destination address and authentication is performed at that destination. Entire inner IP packet is protected by the privacy mechanism, for delivery to the inner IP destination



- For both cases, authentication applies to the ciphertext rather than the plaintext (authentication after encryption)

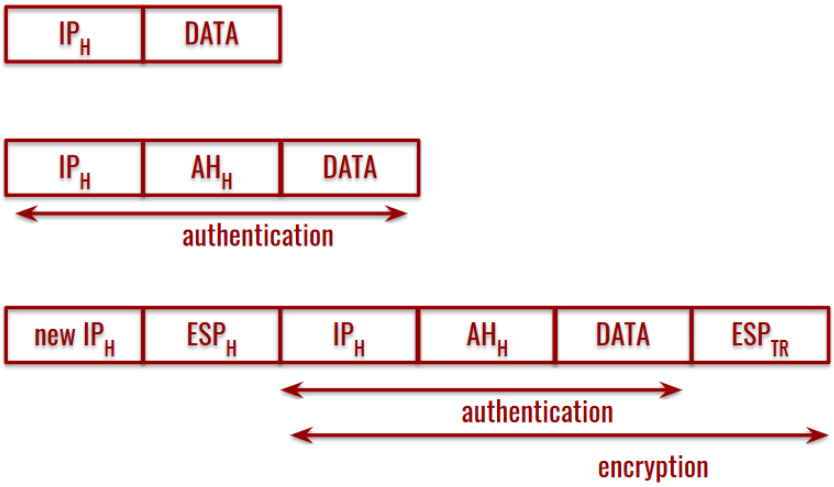
Transport Adjacency

We still have authentication **after** encryption.
This is made by combining ESP and AH, so we combine two SA.s.
The **inner SA is ESP** (no authentication), the **outer SA is AH**. This means that the encryption is applied to the IP payload and so it is added an ESP header, and then all the result is authenticated, so an AH header is added, so the resulting packet is **IP header + ESP**. AH is applied in transport mode, so that authentication covers the ESP + ordinal IP header (except for mutable fields).
The advantage is that with this technique we are **encrypting the data** and we are **authenticating** them, but also some fields of the header like **source and destination IP addresses**.
The disadvantage is that we have overhead (two SAs vs one SA).



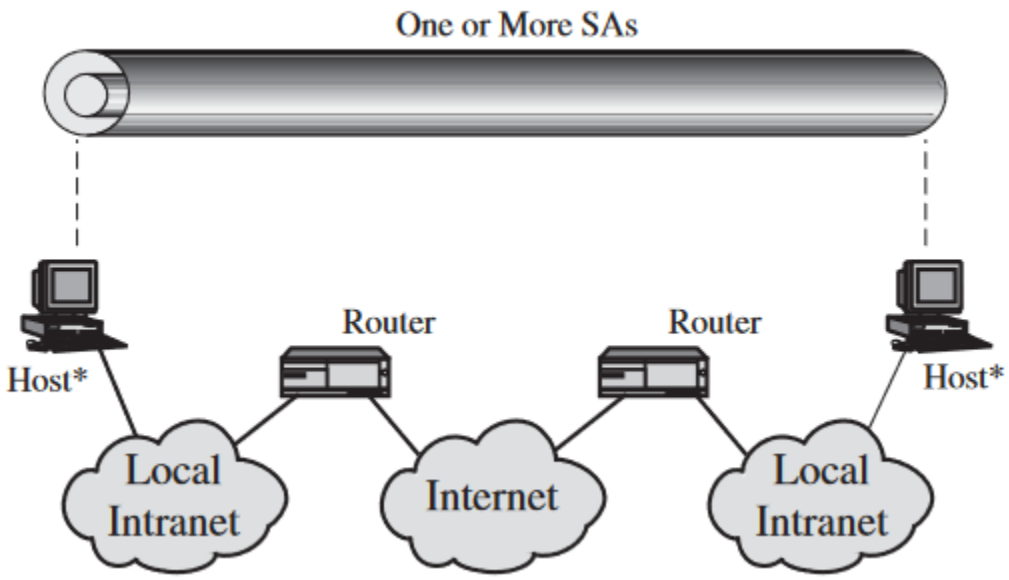
Transport-Tunnel bundle

Authentication is applied **before** encryption, so it is impossible to alter authentication data without decryption. Authentication is applied to the entire IP packet **through an inner AH transport SA** (an AH header is added), and then we encrypt the whole outcoming IP packet by an **outer ESP tunnel SA**, so an ESP header is added (and also a new IP header).



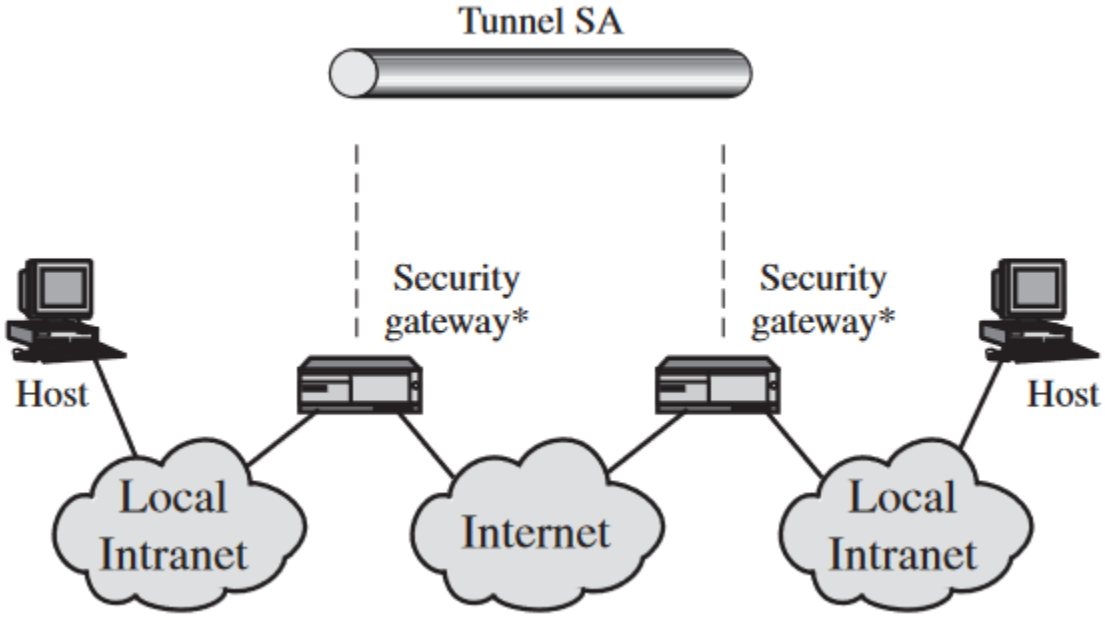
The IPSEC Architecture document lists **four examples of combinations of SAs** that must be supported by compliant IPSEC hosts (e.g. workstation, server) or security gateways (e.g. firewall, router). Here they are:
* = implements IPSEC

Case 1 - User to User



- All security is provided between end systems that implement IPSec.
- For any two end systems to communicate via an SA, they must share the appropriate secret keys.
- Among the possible combinations:
  - AH in transport mode
  - ESP in transport mode
  - ESP followed by AH in transport mode (an ESP SA inside an AH SA)
  - Any one of the preceding, inside an AH or ESP in tunnel mode
- Support for
  - authentication
  - encryption
  - authentication before encryption
  - authentication after encryption

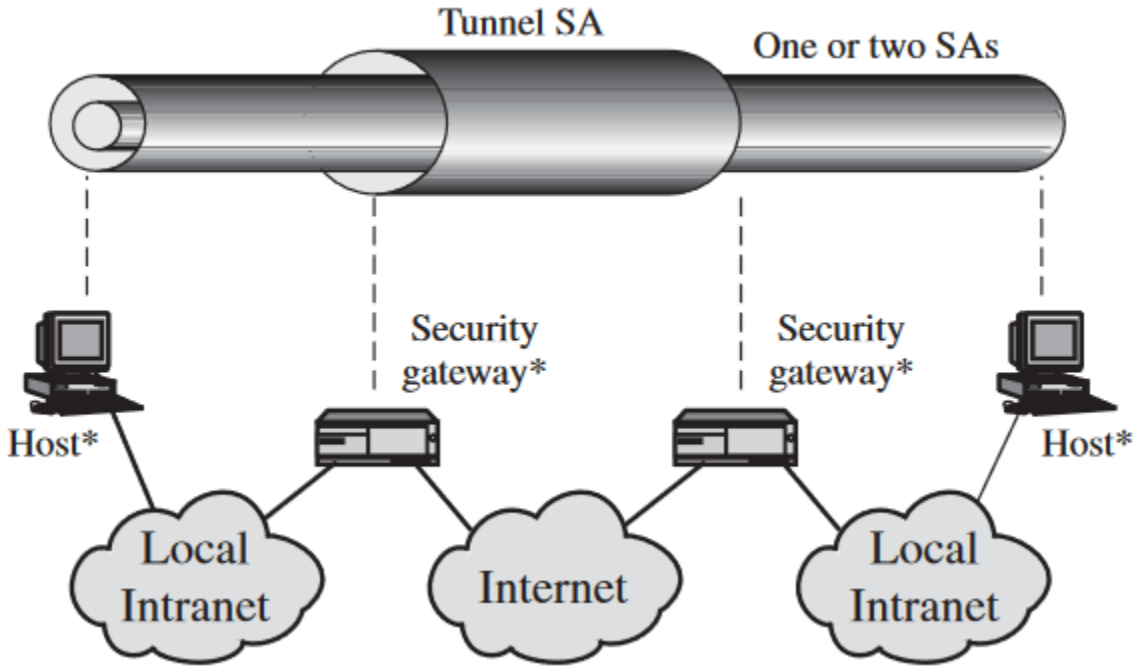
Case 2 - Company to Company



- Security is provided only between gateways (routers, firewalls, etc.) and no hosts implement IPSec (simple virtual private network support)

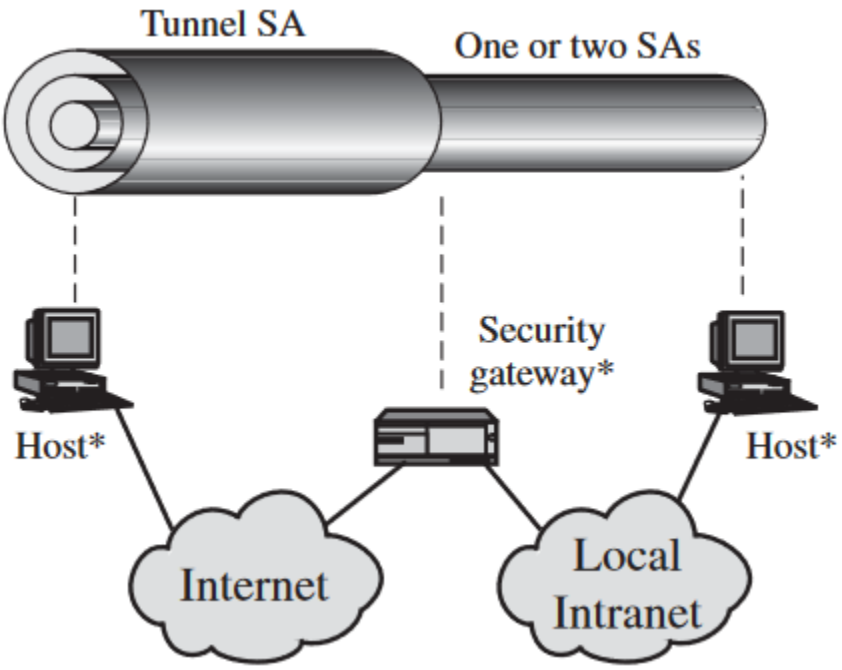
- The security architecture document specifies that only a single tunnel SA is needed for this case. The tunnel could support AH, ESP, or ESP with the authentication option. Nested tunnels are not required because the IPSec services apply to the entire inner packet.

Case 3 - Protected Networks and Users inside the Networks



- Builds on Case 2 by adding end-to-end security. Same combinations discussed for cases 1 and 2 are allowed
- Gateway-to-gateway tunnel provides either authentication or confidentiality or both for all traffic between end systems
- When the gateway-to-gateway tunnel is ESP, it also provides a limited form of traffic confidentiality.
- Individual hosts can implement any additional IPSec services required for given applications or given users by means of end-to-end SAs

Case 4 - Multi Layer Protection



- Provides support for a remote host that uses the Internet to reach an organization's firewall and then to gain access to some server or workstation behind the firewall
- Only tunnel mode is required between the remote host and the firewall. As in Case 1, one or two SAs may be used between the remote host and the local host

Examples

What **Security Association** is better for the following cases?

1. Bob at home wants to print a pdf document on the Laser Printer located in the private network of his corporate.
2. Alice the boss is at home, and needs to save into her Personal Computer in the office document that is **confidential**.

1. Bob needs only **to connect** to the Gateway of the corporate and he needs a **tunnel** of course (host-to-network communication), then the packet can travel in the private network of the corporate in an **unprotected way**. Indeed Bob can just ask for **ESPa** in **tunnel mode** because ESP guarantees that the document is encrypted till it reaches the gateway, and the authentication feature of ESP guarantees that the encrypted document will not be modified, then once the document reaches the gateway it can be sent in clear to Bob's printer since it will travel within the private network of Bob's corporate.

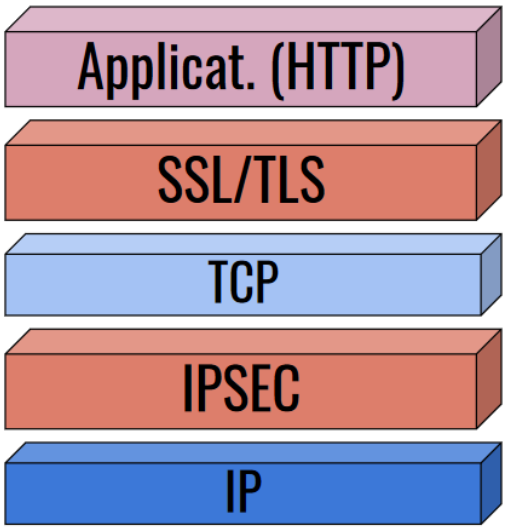
2. She needs **more SA**, so once the packet reaches the gateway and the gateway extracts the payload, this outcome should be **still protected**, so she needs a SA between her and the gateway and a SA between the gateway and her computer in the office. Therefore she needs **tunneling ESP** for the first path (Alice to Gateway) and **ESP** for the second path (Gateway to PC), so she needs two levels of **encryption.**

SSL/TLS

KEY POINTS

- Secure Socket Layer (SSL) provides security services between TCP and applications that use TCP. The Internet standard version is called Transport Layer Service (TLS).
- SSL/TLS provides confidentiality using symmetric encryption and message integrity using a message authentication code.
- SSL/TLS includes protocol mechanisms to enable two TCP users to determine the security mechanisms and services they will use.
- HTTPS (HTTP over SSL) refers to the combination of HTTP and SSL to implement secure communication between a Web browser and a Web server.
- Secure Shell (SSH) provides secure remote logon and other secure client/server facilities.

Transport Layer Security (TLS) and its predecessor **Secure Socket Layer (SSL)** are cryptographic protocols that provide security for communications over networks.



In TCP the sender and the receiver are numbers, i.e. port numbers. When an application is working and it is using the network, a port is associated with it.

With IPSEC we provide security from port to port, now we are **securing a specific application,** a specific port.

In most cases we prefer TLS over IPSEC because IPSEC is a way to secure all the communications between hosts, while TLS secures a specific application. TLS is not invisible to applications, while IPSEC is.

TLS and SSL encrypt the segments of network connections at the Transport Layer providing **end-to-end security**: the scope is to communicate across a network in a way designed to **prevent eavesdropping, tampering** and **message forgery,** since the protocols **provide endpoint authentication** and **communications confidentiality** using cryptography.

Examples of applications using TLS/SSL can be web browsing, email, instant messaging, VoIP.

- **eavesdropping**: the act of secretly listening to private conversation
- **tampering**: the act of altering something secretly or improperly
- **message forgery**: sending of a message to deceive the recipient as to whom the real sender is

Usually in TSL the authentication is **unilateral**: only the server is authenticated, but not vice versa. However TSL also supports **mutual authentication**.
Mutual Authentication requires that the TLS client side also holds a certificate (which is not usually the case in the end-user/browser scenario). In absence of certificates, the TLS-PSK (TSL with pre shared key) or SRP (Secure Remote Protocol) are used to provide strong mutual authentication.

SSL/TLS is used for different scenarios:

- **Key exchange :** RSA, Diffie-Hellman, Elliptic Curves, SRP, PSK
- **Authentication :** RSA, DSA, Elliptic Curve Digital Signature Algorithm
- **Symmetric Ciphers :** RC4, 3DES, AES, DES
- **Cryptographic Hash Functions :** HMAC-MD5, HMAC-SHA

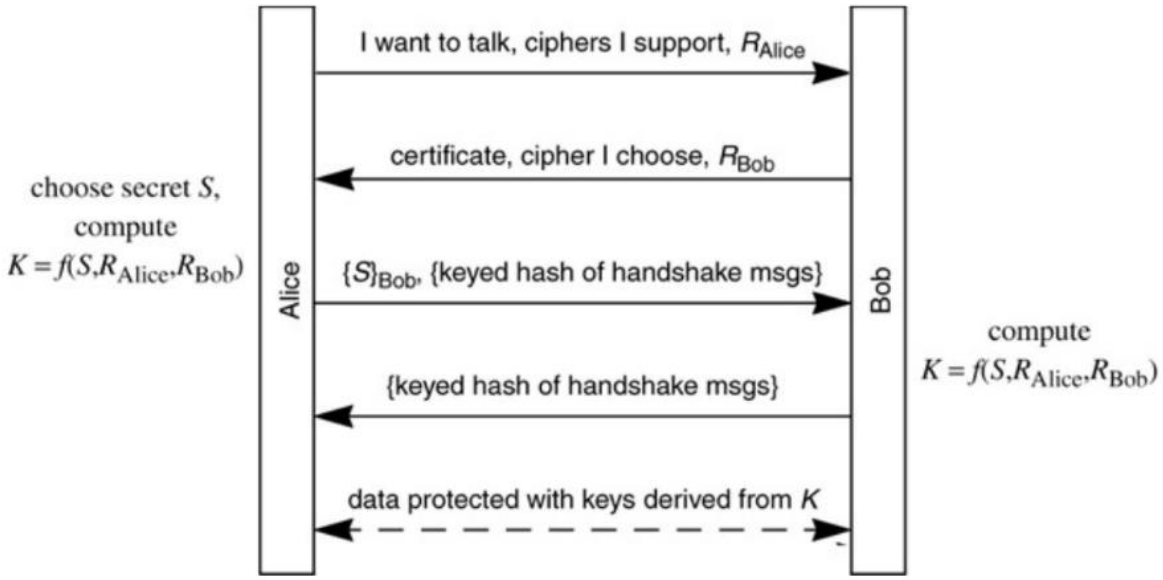
SSL/TLS involves **three basic phases:**

1. Peer negotiation for algorithm support
2. Key exchange and authentication (here it is mainly used public key cryptography)
3. Symmetric cipher encryption and message authentication

This is **how SSL/TLS works**:

1. Client and server negotiate a stateful connection by using a **handshaking procedure**. During handshake, client and server agree on various parameters used to establish connection's security and exchange some random numbers.
2. Handshake begins when the client connects to a TLS-enabled server requesting a secure connection and presents a list of supported ciphers and hash functions.
3. From this list, the server picks the strongest cipher and hash function that it also supports and notifies the client of the decision.
4. Server **sends back its identification** in the form of a digital certificate **X.509.**
5. Client may contact the CA and confirm that the certificate is authentic and not revoked before proceeding.
6. For generating session keys used for **secure connection**, for instance when using RSA for key exchange the client may **encrypt a secret number** (S) with the server's public key (PbK), and send the result to the server.

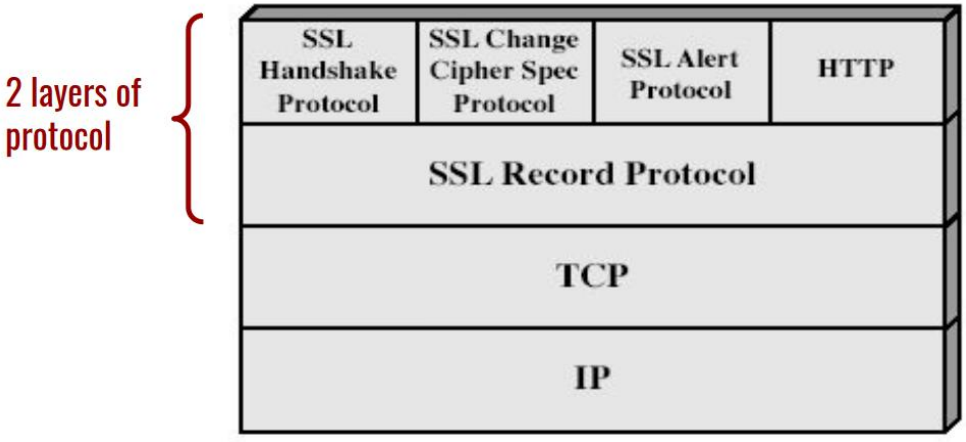
7. From the secret number, **both parties generate key material** for encryption and decryption.
8. This concludes the handshake and **begins the secured connection**, which is encrypted and decrypted with the key material until the connection closes.
9. If any one of the above steps fails, the TLS handshake fails, and the connection is not created.



Remark. This is just a basic idea. TLS can generate different messages based on specific cipher suite in use.

SSL Architecture

SSL - Secure Socket Layer, is a **transport layer security service**. It uses TCP to provide a reliable end-to-end service.
SSL has **two layers of protocols**.



The **first layer** is the one which performs the **setup of the connections**, so in this phase all the useful parameters in order to **secure** the conversation are exchanged.
The **second layer** is the **Record Protocol**, which acts after the setup session, in which we actually encrypt and decrypt the real data of the conversation.

- **SSL Session :** in the result of an handshaking, the two parties establish some parameters and all such parameters describe a **session**. Within the session you can establish several **connections** using the same parameters.
  We reuse parameters cause you do not want to start a new handshaking phase for every connection.
  Thus, an SSL session is a **logical association** between client and server.
- **SSL Connection :** it is just an **exchange of messages** within a session, so more connections can take place in one single session and it is made to save some computation.

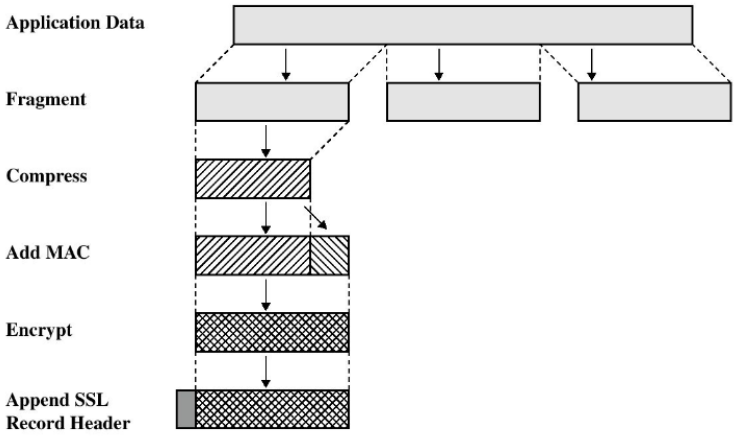
Between any pair of parties there may be multiple secure connections. (there may also be multiple simultaneous sessions between parties, but this feature is not used in practice).
Several states are associated with each session: once a session is established, there is a **current operating state** for both read and write (i.e. receive and send). **During** handshake protocol, pending read and write states are created. **After** the conclusion of the handshake protocol, the pending states become the current states.

SSL Record Protocol

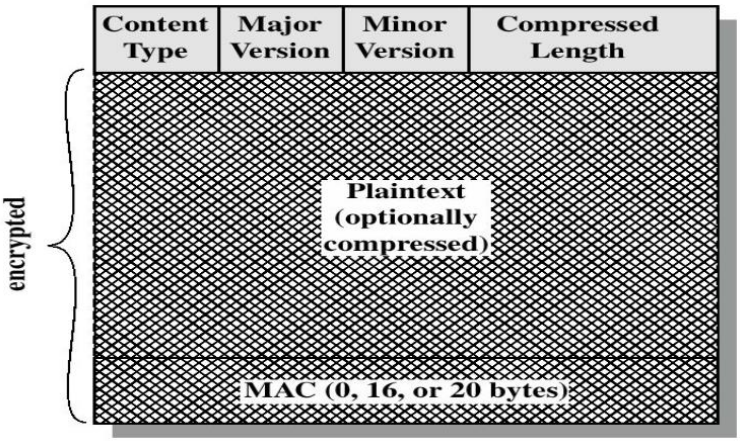
This protocol offers two main services:

- **Confidentiality :** using symmetric encryption with a shared secret key defined by handshake protocol (maybe using RC2-40, DES-40, DES, 3DES, RC4-40, RC4-128). A message is compressed before encryption.
- **Message Integrity :** using MAC with shared secret key. Similar to HMAC but with different padding.

This is the scheme of **Record Protocol** which takes place after the handshaking phase, so once all the MACs and cipher specifications have been decided.
The application data is **fragmented** and **compressed**, then **authenticated** (MAC) and **encrypted**.
Then, the **SSL Record Header** is appended.

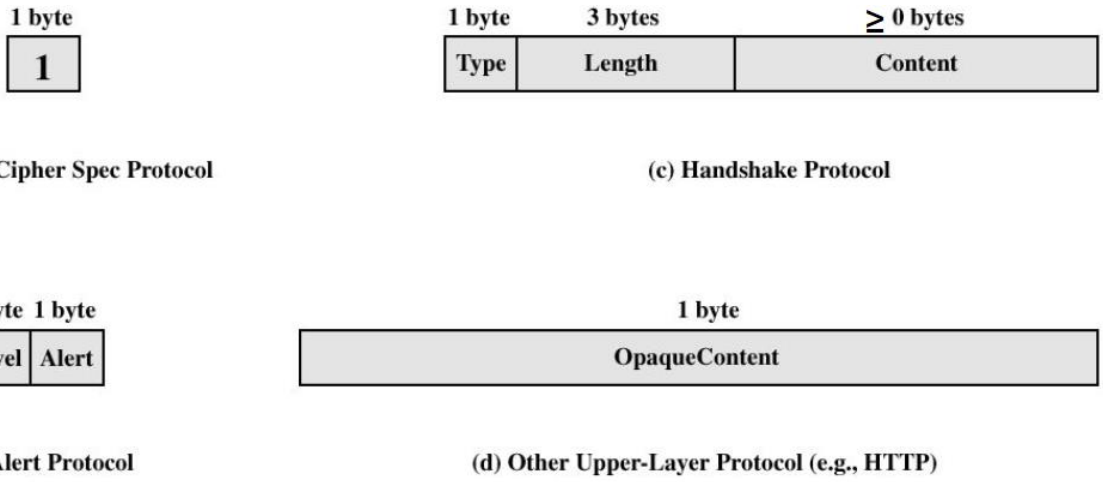


A **single SSL Record** of the record protocol is like this:
Content Type (8 bits) is used to process the enclosed fragment (cipher specifications ecc).
Major Version (8 bits) indicates the major version of SSL in use, as the
Minor Version (8 bits) that indicates the minor one.
Compressed Length (16 bits) is the length in bytes of the plaintext fragment (or compressed fragment if compression is used).



The **encrypted part** is composed by the plaintext and the MAC.

The **SSL Payload** is the normal information transferred by the protocol.



There are **four kinds of payload**:

- **Change Cipher Spec Protocol :** it is just one byte, it acts like a signal saying “ok,we are done with our handshaking, now let’s start the game by encrypting/decrypting and authenticating messages”. It is a single message, and needs to cause the pending state to be copied into the current state, which updates the cipher suite to be used on this connection. Usually sent **after handshaking**.
- **Alert Protocol** : composed by two bytes, used to specify alerts like fatal or warnings.
- **Handshake Protocol:** one byte of Type, three bytes of length and then the content.
- **Other Upper-Layer Protocol:** one byte of opaque content.

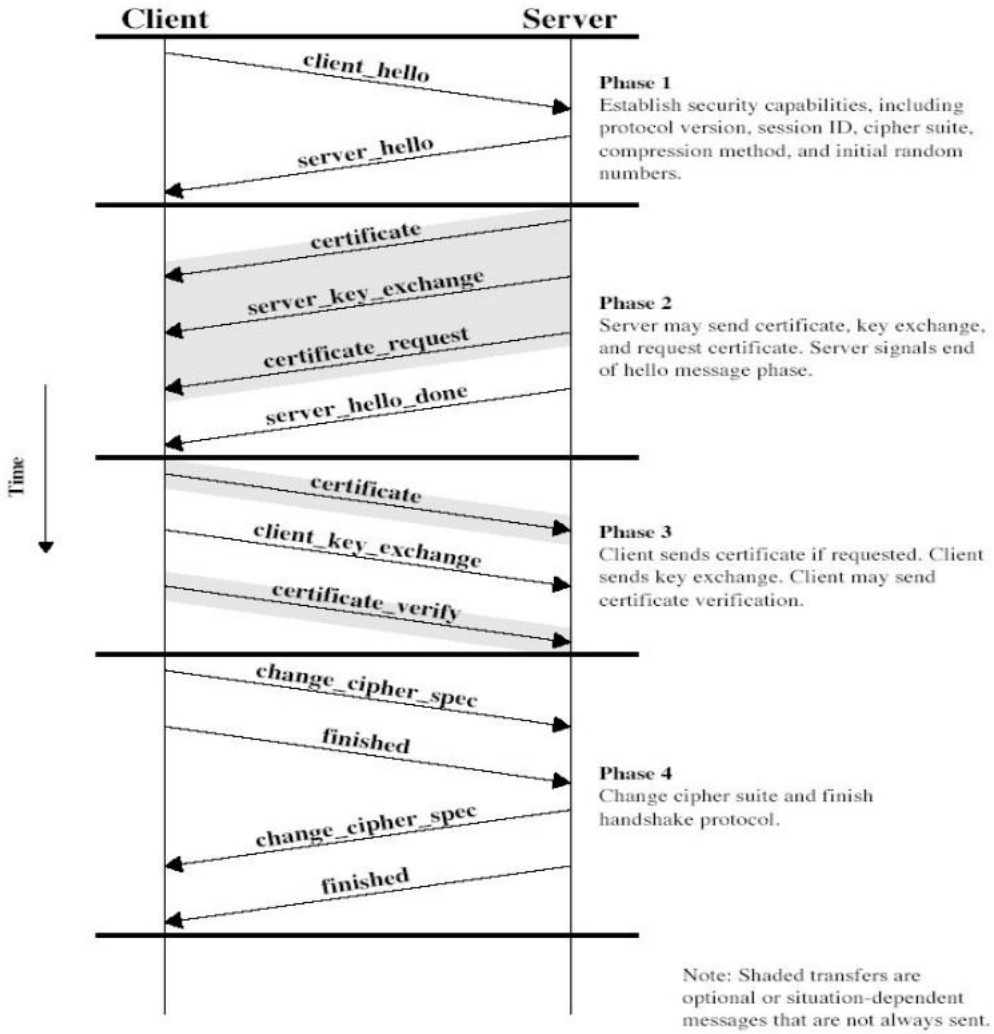
SSL Handshake Protocol

The **Handshake Protocol** is the most important part of SSL. It is complex and somewhat long, indeed when people, normal users, are connecting to aTLS enabled website, they can feel a small delay that is related to the handshaking phase.
This protocol allow server and client to **authenticate each other**, negotiate **encryption** and **MAC** algorithms and also the **keys** to be used.

It is composed by **Four Phases:**

1. **Hello Phase :** client presents himself to the server giving him some parameters such as the SSL/TLS used version, the list of supported ciphers, the session ID, the compression algorithms and so on. Client here is basically **determining security capabilities**.
2. **Server Hello :** server responds to the client, deciding for him the algorithms to use. **Server sends certificate and asks for certificate** to client and then the session keys exchange starts.
3. **Client sends information :** client offers to the server the requested certificate and completes the key exchange. Then the client confirms the end of the handshake.

4. **End of handshake :** also the server confirms the end of the handshake.



Several algorithms are used for key exchange (phases 1, 2): RSA (session key is encoded with server public key), Diffie-Hellman (in several versions: fixed, ephemeral, anonymous), Fortezza.

- **Fixed Diffie-Hellman (or static DH)**
  The original Diffie-Hellman Key Exchange is **vulnerable to man in the middle attack** if parties are not authenticated.
  In the case of Fixed DH the server's certificate contains **DH public parameters signed by the Certification Authority** (CA) and the client provides its DH public key parameters with a certificate (if client authentication is required) or in a key exchange message. The client can perform static (called static-static DH) or ephemeral DH (more common, called static-ephemeral DH).
  This method results in a **fixed secret key** between two peers, based on the DH calculation using the fixed public keys.
- **Ephemeral Diffie-Hellman**
  Used to create **one time secret keys** (ephemeral). In this case, the DH public keys are

exchanged and signed using the sender's private RSA or DSS key.
The receiver can use the corresponding public key in order to verify the signature, and certificates are used to authenticate the public keys. This can be considered the most secure one since the **authenticated key is temporary**.

- **Anonymous Diffie-Hellman**
  It is the base DH algorithm, so **without authentication**, where each party sends its public DH parameters to the other.
  However this approach is vulnerable to MITM attacks and so it is not allowed anymore since TLS 1.2.

So, for a fast recap..
If the server embeds fixed params in the certificate and the client chooses its own params is **static ephemeral DH scenario**.
If we have fixed params on both sides it is a **static DH scenario**, where we have authentication and forward secrecy, but we always use the same keys.
If for each session both sides use different params it is an **ephemeral DH scenario**, that is the most secure one.
The **anonymous DH scenario** is when we do not have authentication, and so it is susceptible to a MITM attack. So why use it? If you **do not have a certificate.**

Attacks against SSL/TLS

Downgrade Attack - An adversary cheats the server and other clients in order to use an older protocol, so in this way he can use older algorithms parameters that he can manipulate and that are no more secure, in such a way that he can modify the traffic flow in order to accomplish a man-in-the-middle.

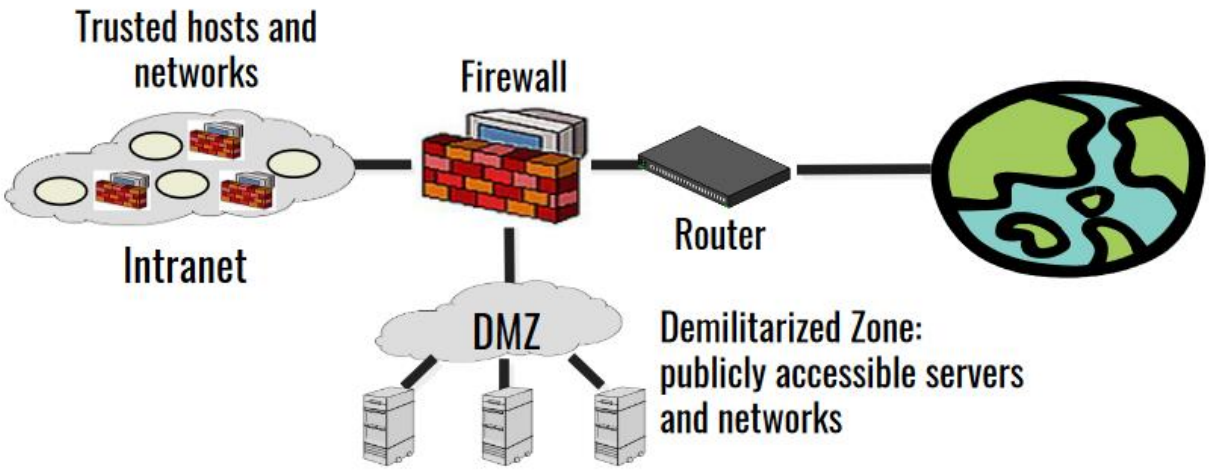
Heartbleed Attack - Heartbeat was an extension of theTLS protocol that offers a way to keep alive secure communication without needing to restart the connection each time based on exchange of messages.
But the older version allocates memory buffer(server-side) for the message to be returned based on the length field in the requesting message, without regard to the actual size of the payload, so an adversary cheats on the length field, so he will receive a huge amount of data from the secure server.

Firewall

KEY POINTS

- A firewall forms a barrier through which the traffic going in each direction must pass. A firewall security policy dictates which traffic is authorized to pass in each direction.
- A firewall may be designed to operate as a filter at the level of IP packets, or may operate at a higher protocol layer.

The idea is to **separate the local network from the internet** in such a way that every data entering the local network will be analyzed by the firewall.



In most cases a firewall links an internal network to the external world.
Various scopes for the firewall: it limits the inbound and outbound traffic, controls and monitors access to services, hides the internal network to the external world and so on.

Firewalls can be categorized in **three types,** each one working on a different level of the network:

- **Packet Filtering (**or session-filtering router)**:** filtering is done by **inspecting headers** of the packets (even payloads in some cases). It is usually stateless, sometimes stateful (session).
- **Proxy Gateway :** the firewall acts like a gateway, in the sense that all incoming traffic is directed to the firewall and all outgoing traffic appears to come from the firewall. It can be categorized in **circuit level,** where the firewall works at lower level, so it is invisible to

application level, and **application level,** where every application has its own proxy firewall.
- **Personal Firewall :** every PC has a firewall.

Packet Filtering

For each packet, the firewall decides whether to allow it to proceed: decision must be made on a **per-packet** basis.
Firewalls can operate in stateless or stateful mode. **Stateless** means that every packet is analyzed independently from an other, **stateful** means that the firewall maintains some information about the packets transiting through it.
Usually the **header of the packet** is analyzed, in order to inspect the **IP source** of the packet, but it can be possible that further analysis is needed, so the firewall may inspect the **payload** of the packet in order to get TCP port and take further decisions.
So to summarize, packet filtering is based on **filtering IP numbers and TCP ports**.

Being **stateful** means that the firewall keeps information about some session, for example a TCP session. In such a way the firewall is able to understand what every packet means within that session, therefore we can apply a kind of **session filtering**. It means that the firewall takes different decisions depending on the packet in the context of a session. This kind of firewall must be placed at a higher **level** with respect to IP level.

An example of **filtering packets** can be analyzing all packets coming from 192.168.4.*, where * means any number.
If we know that a certain subnet is dangerous, let's say subnet 10.25.68.0/24, we can add a **rule**, writing something like "10.25.68.* BLOCK" in order to block all the traffic coming from that subnet.

Packet filtering is **vulnerable to IP Spoofing** since the attacker can change its IP bypassing the rules of the firewall.
A well known attack against firewalls is **fragmentation attack**. The idea is simple: fragment a packet into two sub packets such that they seem harmless alone, but once reassembled on the upper layer they constitute malicious fragments.
Also **DOS** attacks can be performed.

So some **weaknesses of packet filters** can be:
- Do not prevent application-specific attacks: e.g. if there is a buffer overflow in URL decoding routine, firewall will not block an attack string
- No user authentication mechanism
- Vulnerable to TCP/IP attacks such as spoofing. A solution can be to set a list of addresses for each interface
- Security breaches due to misconfiguration

Here are some examples of rule tables:

Rule Set A

action	ourhost	port	theirhost	port	comment
block	*	*	SPIGOT	*	we don't trust these people
allow	OUR-GW	25	*	*	connection to our SMTP port

Rule Set B

action	ourhost	port	theirhost	port	comment
block	*	*	*	*	default

Rule Set C

action	ourhost	port	theirhost	port	comment
allow	*	*	*	25	connection to their SMTP port

Rule Set D

action	src	port	dest	port	flags	comment
allow	{our hosts}	*	*	25		our packets to their SMTP port
allow	*	25	*	*	ACK	their replies

Rule Set E

action	src	port	dest	port	flags	comment
allow	{our hosts}	*	*	*		our outgoing calls
allow	*	*	*	*	ACK	replies to our calls
allow	*	*	*	>1024		traffic to nonservers

A. Inbound mail is allowed (port 25 is for SMTP incoming), but only to a gateway host. However, packets from a particular external host, SPIGOT, are blocked because that host has a history of sending massive files in e-mail messages.

B. This is an explicit statement of the default policy. All rulesets include this rule implicitly as the last rule.

C. This ruleset is intended to specify that any inside host can send mail to the outside. A TCP packet with a destination port of 25 is routed to the SMTP server on the destination machine. The problem with this rule is that the use of port 25 for SMTP receipt is only a default; an outside machine could be configured to have some other application linked to port 25. As this rule is written, an attacker could gain access to internal machines by sending packets with a TCP source port number of 25.

D. This ruleset achieves the intended result that was not achieved in C. The rules take advantage of a feature of TCP connections. Once a connection is set up, the ACK flag of a TCP segment is set to acknowledge segments sent from the other side. Thus, this ruleset states that it allows IP packets where the source IP address is one of a list of designated internal hosts and the destination TCP port number is 25. It also allows incoming packets with a source port number of 25 that include the ACK flag in the TCP segment. Note that we explicitly designate source and destination systems to define these rules explicitly.

E. This ruleset is one approach to handling FTP connections. With FTP, two TCP connections are used: a control connection to set up the file transfer and a data connection for the actual file transfer. The data connection uses a different port number that is dynamically assigned for the transfer. Most servers, and hence most attack targets, use low-numbered ports; most outgoing calls tend to use a higher-numbered port, typically above 1023. Thus, this ruleset allows

— Packets that originate internally

— Reply packets to a connection initiated by an internal machine

— Packets destined for a high-numbered port on an internal machine

This scheme requires that the systems be configured so that only the appropriate port numbers are in use.

And more:

The following filtering rules, defined as access control lists (ACLs) on a CISCO router, allow a user to FTP from any IP address to the FTP server at 172.168.10.12

```
access-list 100 permit tcp any gt 1023 host 172.168.10.12 eq 21
access-list 100 permit tcp any gt 1023 host 172.168.10.12 eq 20
! Allows packets from any client to the FTP control and data ports
access-list 101 permit tcp host 172.168.10.12 eq 21 any gt 1023
access-list 101 permit tcp host 172.168.10.12 eq 20 any gt 1023
! Allows the FTP server to send packets back to any IP address with TCP ports > 1023

interface Ethernet 0
 access-list 100 in ! Apply the first rule to inbound traffic
 access-list 101 out ! Apply the second rule to outbound traffic
!
```

Anything not explicitly permitted by the access list is denied!

Iptables

It is an interface for dealing with the firewall on linux, that is called **netfilter**.
Iptables is not the firewall, it is how we interact with it, to set up, maintain and inspect the tables of IPv4 packet filter rules in the Linux kernel.
Linux is moving from iptables in favor of **nftables**. Iptables can be translated into nfilters rules using iptables-translate.
Iptables works at **packet level**, not at application level.

Several different **tables** may be defined.
Each table contains a number of **built-in chains** and may also contain **user-defined chains**.
A chain is a **list of rules**, processed in a sequential way, which determine whether a packet has to be accepted or dropped.
Each rule specifies criteria for a packet and an associated **target**, namely what to do with a packet that matches the patterns.

There exist some kinds of **built-in chains:**

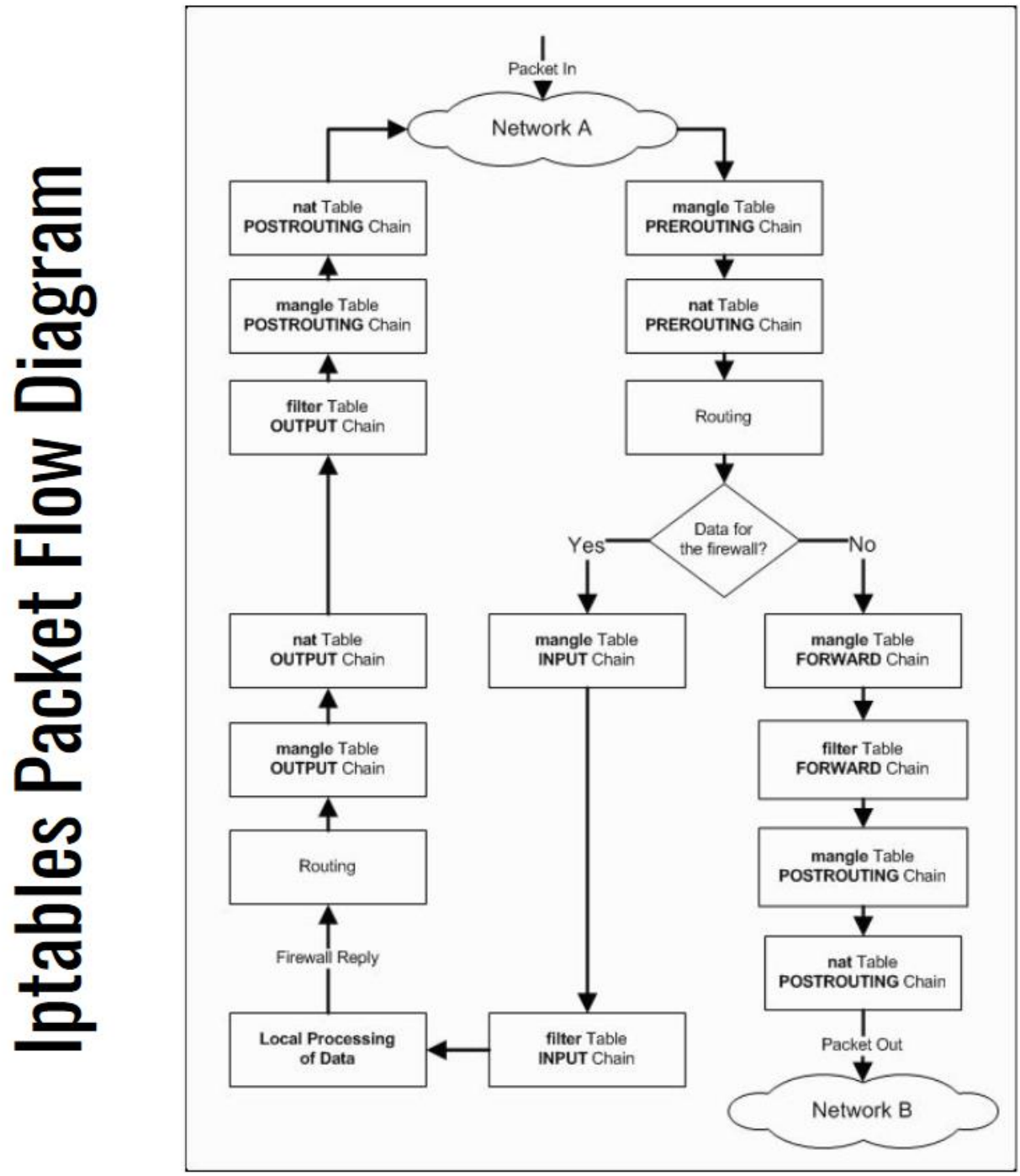
- **Prerouting:** Packets will enter this chain before a routing decision is made.
- **Input:** Packet is going to be locally delivered. This is a list of rules that consider **incoming packets**.
- **Forward:** All packets that have been routed and were not for local delivery will traverse this chain. This is a list of rules that consider all packets that **were routed to us but not delivered** for us so they just have to be forwarded.
- **Output:** Packets sent from the machine itself will be visiting this chain. This chain deals with **outgoing packets**.
- **Postrouting:** Routing decision has been made. Packets enter this chain just before handing them off to the hardware.

When we are implementing a personal firewall then Input and Output chains are meaningful, instead Forward chains are not important. On the contrary, when IPtables is working on gateway, then Forward chains are the most important.

As said, each rule specifies criteria for a packet and a **target**. If the packet does not match a rule, the next rule is then examined. If it matches, then the next rule is specified by the value of the target.
The most **common targets** are:

- **Accept** : let the packet through
- **Drop :** drop the packet on the floor
- **Queue :** pass the packet to userspace (what actually happens depends on a queue handler, included in all modern linux kernels)
- **Return :** stop traversing this chain and resume at the next rule in the previous (calling) chain

The **flow diagram** below represents **how an iptable works**.
Most firewalls put rules in the **filter table.** Some data are specific for the firewall, some others (most common) just have to transit from **network A** to **network B.**



Iptables commands and examples

(often asked in the exam) (When you reboot your machine you lose iptables rules, so must save them)
From the **command line** give the command > *iptables*. Then some commands are available and here reported. (Exclamation point *!* is often used by iptables for negation)

- -A (--append) chain rule-specification: append a new rule
- -L (--list) [chain]: list rules
- -S [chain]: print rules
- -F [chain]: flush (delete) rules
- -D chain rulenum: delete a specific rule in a chain
- -P chain target: set default policy for a chain

ADDING A NEW RULE

Command syntax for appending a new rule into a chain:

> IPTABLES -t TABLE -A CHAIN -[i|o] IFACE -s x.y.z.w -d a.b.c.d -p PROT -m state --state STATE -j ACTION

Rules use:

- PACKET MATCHING TABLE = nat | filter | ...
- ORIGIN OF CONNECTION/PACK (CHAIN). = INPUT (I) | OUTPUT (O)| FORWARD (F) | ...
- NETWORK INTERFACE (IFACE) = eth0 | eth1 | ppp0 (network adapter)
- PROTOCOL (PROT) = tcp | icmp | udp .....
- STATE OF THE CONNECTION (STATE) = NEW | ESTABLISHED | RELATED .....

Based on the rules there is an action:

- ACTION ON THE PACKET = DROP | ACCEPT | REJECT | DNAT | SNAT .....

Input and output interfaces can be the ethernet card, wireless card and similar.

Example 0

Assume *eth0* interface of a router to the public internet.
Remark: packets are discarded with no reply to the sender; in this way the firewall protects against flooding attacks and does not provide information for attacks based on “port scanning”.

- Block all incoming traffic: iptables -A FORWARD -i eth0 -j DROP

```
> iptables -A FORWARD -i eth0 -j DROP
> iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  anywhere             anywhere

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
> iptables -D FORWARD 1 # to remove it
```

Brief analysis:
-A FORWARD : the command -A is used to append a new rule. The argument of this command is a **chain**, the origin of connection/packet. FORWARD is a valid argument, since **forward** is one kind of built-in chains (see some pages above).
More specifically, *forward = all packets that have been routed and were not for local delivery will traverse this chain.*
So we are now selecting all the packets that transit into this chain.

-i eth0 : the command -i stands for “input”, and the argument is *eth0.* So we are selecting all the incoming packets from the interface *eth0*.

-j DROP : the command -j takes as an argument the action we want to perform (drop, accept, reject, etc.). When we **drop**, we do not send back any response of the block of the incoming traffic.

Example 00

- Accept pck from outside if they refer to a TCP connection started within the network

  iptables -A FORWARD -i eth0 -m state --state ESTABLISHED -j ACCEPT

  Remark: state “ESTABLISHED” allows to decide whether the connection originated from the inside or the outside; ESTABLISHED information is stored in the IPTABLES.

Example 1

Allow firewall to **accept TCP packets for routing** when they enter on interface **eth0** from **any IP address** and are destined for an IP address of 192.168.1.58 that is reachable via interface **eth1**. The source port is in the range 1024 to 65535 and the destination port is port 80 (www/http).

```
iptables -A FORWARD -s 0/0 -i eth0 -d 192.168.1.58 -o eth1 -p TCP
    --sport 1024:65535 --dport 80 -j ACCEPT
```

The packet is not for us, but **for a machine in the network**, so it must go in the **FORWARD** chain.
-s 0/0 : indicates that the **source** of our packet can be any IP address (0/0), so our packet can come from anywhere.
-d 192.168.1.58 : specifies the **destination** ip address where to forward any incoming packet.
-o eth1 : means that once coming through this rule, the packet must be routed toward the *eth1* interface to be sent to the specified ip address.
-p TCP : the command -p specifies the protocol (as argument) that can be tcp, udp, icmp, etc.
--sport and **--dport** are indicating the source port and the destination port.
-j ACCEPT : is the action, what to do with the packets.

Example 2

Allow the firewall to send ICMP echo-requests (pings) and in turn accept the expected ICMP echo-replies.

```
iptables -A OUTPUT -p icmp --icmp-type echo-request -j ACCEPT
iptables -A INPUT -p icmp --icmp-type echo-reply -j ACCEPT
```

Example 3

- accept at most 1 ping/second

```
iptables -A INPUT -p icmp --icmp-type echo-request -m limit --limit
    1/s -i eth0 -j ACCEPT
```

- limiting the acceptance of TCP segments with the SYN bit set to no more than five per second

```
iptables -A INPUT -p tcp --syn -m limit --limit 5/s -i eth0 -j ACCEPT
```

This is useful to limit incoming packets to avoid a DOS attack

Example 4

- Allow the firewall to accept TCP packets to be routed when they enter on interface eth0 from any IP address destined for IP address of 192.168.1.58 that is reachable via interface eth1. The source port is in the range 1024 to 65535 and the destination ports are port 80 (www/http) and 443 (https).

```
iptables -A FORWARD -s 0/0 -i eth0 -d 192.168.1.58 -o eth1 -p TCP --sport 1024:65535
    -m multiport --dports 80,443 -j ACCEPT
```

- The return packets from 192.168.1.58 are allowed to be accepted too. Instead of stating the source and destination ports, you can simply allow packets related to established connections using the -m state and --state ESTABLISHED options.

```
iptables -A FORWARD -d 0/0 -o eth0 -s 192.168.1.58 -i eth1 -p TCP -m state --state
    ESTABLISHED -j ACCEPT
```

Notice the difference with **Example 1**: here we have that **-m multiport** indicates that multiple ports can be used as arguments of **--dports**.

For example in the second rule, the source **-s** could be my web server, while we admit any destination **-d 0/0**.

Example 5

- allow DNS access from/to firewall

```
iptables -A OUTPUT -p udp -o eth0 --dport 53 --sport 1024:65535 -j ACCEPT
iptables -A INPUT -p udp -i eth0 --sport 53 --dport 1024:65535 -j ACCEPT
```

DNS uses UDP protocol, so **-p udp** is used.
Notice that the rules are for both send and reply.
Here there is a **weakness**: we allow replies by we can't say if a reply is related to an earlier request. This is an **UDP weakness**.

Example 6 - SSH

- allow www & ssh access to firewall

```
iptables -A OUTPUT -o eth0 -m state --state ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -p tcp -i eth0 --dport 22 --sport 1024:65535 -m state --state NEW -j ACCEPT
iptables -A INPUT -p tcp -i eth0 --dport 80 --sport 1024:65535 -m state --state NEW -j ACCEPT
```

The **second rule** says that if the firewall that is running the web server and SSH server receives a request from a client that wants to interact with it, then **accept**.

Example 7

- allow firewall to access the Internet
  - enables a user on the firewall to use a Web browser to surf the Internet. HTTP traffic uses TCP port 80, and HTTPS uses port 443

```
iptables -A OUTPUT -j ACCEPT -m state --state
    NEW,ESTABLISHED,RELATED -o eth0 -p tcp -m multiport --dports
    80,443 --sport 1024:65535

iptables -A INPUT -j ACCEPT -m state --state ESTABLISHED,RELATED -i
    eth0 -p tcp
```

Port 80 is for http, port 443 is for https.

Example 8

- allow home Network to access firewall
  - in the example, eth1 is directly connected to a home network using IP addresses from the 192.168.1.0 network. All traffic between this network and the firewall is simplistically assumed to be trusted and allowed.

```
iptables -A INPUT -j ACCEPT -p all -s 192.168.1.0/24 -i eth1
iptables -A OUTPUT -j ACCEPT -p all -d 192.168.1.0/24 -o eth1
```

Incoming and outcoming messages from/to the firewall are from our subnetwork.

Bastion Host

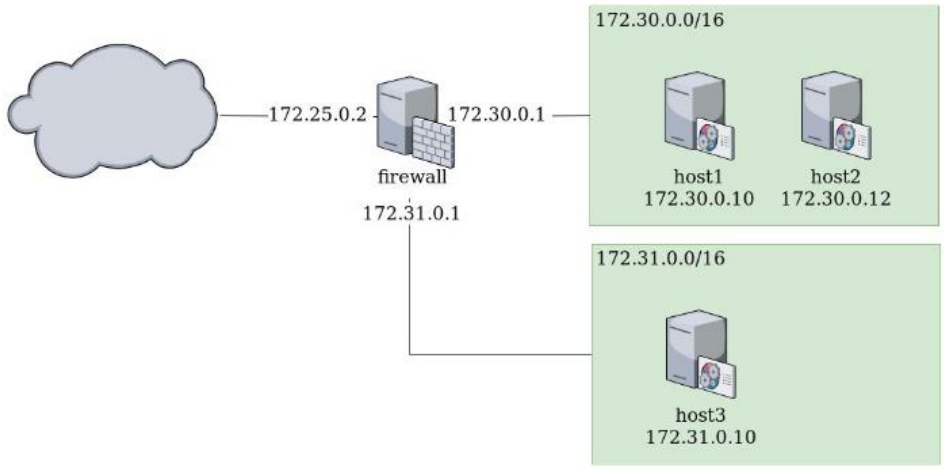
A **bastion host** is a special-purpose computer on a network specifically designed and configured to withstand attacks.

1. unique host that is reachable from the Internet
2. massively protected host
3. secure operating system (hardened or trusted)
4. no unneeded software, no compilers & interpreters
5. proxy server in a insulated environment (chrooting)
6. read-only file system
7. process checker
8. integrity file system checker
9. small number of services and no user accounts
10. untrusted services have been removed
11. saving & control of logs
12. source-routing disabled

A **Bastion Host** is an **hardened system** implementing application level gateway (so needs a proxy firewall for each application) behind packet filtering.
It uses only trustable operating systems and runs only few applications (so all non essential services are turned off), so each proxy only supports a subset of application's command, and the traffic is analyzed, and the access to the disk is restricted, too.

Examples and Exercises

Exercise 1 - Write the rule that drops all tcp connections from host3 (172.31.0.10) to host1 (172.30.0.10) on port 8080.



>iptables -A FORWARD -p tcp -s 172.31.0.10 -d 173.30.0.10 --dport 8080 -j DROP

Mixed Exercises

EXAM 2021 Jan 10th

Q1 True or false

- Q1.1: HMAC is a cryptographic hash function →**FALSE**
- Q1.2: SHA-3 is not vulnerable to the birthday paradox →**FALSE**
- Q1.3: Diffie-Hellman Key Exchange (DHKE) is based on a trapdoor function →**FALSE**
- Q1.4: RSA is based on a trapdoor function →**TRUE**
- Q1.5: RSA is insecure and should not be used →**FALSE**
- Q1.6: A password using 14 chars is always more secure than a password using 12 chars →**FALSE ***
- Q1.7: Kerberos v4 is based mainly on asymmetric ciphers →**TRUE**
- Q1.8: AES can encrypt only a fixed amount of bits →**TRUE ****
- Q1.9: 3-DES can encrypt an arbitrary amount of bits →**FALSE**
- Q1.10: Static DHKE in SSL is secure against a MITM attack →**FALSE**

***Q1.6 : if the 12 chars password is randomly generated and the 14 chars password is something like “ciao1234567890” then we may consider the 12 chars password more secure. However, if we consider the security only on a computational point of view, we can say that to crack a 14 chars password an attacker needs 2^{112} attempts, while the other only 2^{96}.

**Q1.8 : AES can only encrypt a fixed amount of bytes, that is 16 bytes. To encrypt variable length messages it must use an operational mode. So in some sense AES can encrypt variable length messages.

Q2.3 Discuss in detail how RSA can be used for implementing a digital signature scheme. Also, present the main ideas behind one standard of your choice for an RSA-based digital signature scheme.

To implement a digital signature scheme using RSA consists in **encrypting a message with the private key** so that when you send the message to Alice, she can **decrypt it by using your public key**. In this way she can be sure that you sent the message.
However this method does not ensure non-repudiation and an attacker can intercept the message and send it again in another moment. To cope with this problem it would be good to send a timestamp or a challenge within the message, so that the attacker cannot reuse it.

Q4.1 Suppose that a web server is publishing a file that you want to download. To Help you trust the content of the file, the web server is also publishing the SHA-3 checksum computed over the content of the file. Hence, you can easily locally compute the SHA-3 checksum of the file (after downloading it) and compare the obtained hash value with the one published by the web server. Discuss the security of this approach with respect to **data integrity** and **data origin**.

By applying the published SHA-3 checksum over the file you can compare your result with the server's one: if the two results are equal you can be sure that the file integrity has not been compromised. What about data origin? Can we be sure that the file truly comes from the server? **No, we don't have data origin**, because no secret key is shared between client and server, so a MITM could stop the download of the file and the hash and send another file with its own hash, resulting valid for the user.

Q4.2 Suppose that a user chooses a secret key by randomly typing 16 times on a keyboard. Assume that each character typed in can be represented using 1 byte. Discuss the security of this secret key: e.g., is it secure as a 128-bit random number? Is there any benefit with respect to a 128-bit random number when using the secret key as a password for authentication?

From a computational point of view, both keys require at most 2^{128}attempts to be brute-forced. However some analysis can be done: a human randomly typing on a keyboard can produce some patterns, mayber typing one letter with his left hand and the next letter with the right hand, or maybe some special characters can be omitted, lowercase and uppercase letters can follow a scheme and so on. All these things may help to reduce the computational effort when trying to guess or brute force the password, and this makes the human generated password more weak.

Q4.3 Suppose that Bob knows the public key pk(A) of Alice and that the private key pr(A) of Alice has not been compromised. Alice and Bob agreed on the special secret message X = "Bazinga!". For one-way authentication,Alice sends to Bob Encpr(A)(X). Bob decrypts the message using the public key pk(A) of Alice, checks the validity of the message, and authenticates Alice. Discuss the security of the protocol and possible improvements.

In this case Bob can be sure that Alice herself sent the message, since it was encrypted with Alice's private key. However this protocol may be susceptible to **replay** attacks: an attacker can intercept the message and send it again whenever it wants. To avoid this, a timestamp or a challenge can be added to the protocol, so that they can be sure that no one is reusing messages.
Another problem can be that an active eavesdropper could intercept the message, decrypt it with Alice's public key and do whatever he wants. In this situation a possible solution for Alice could encrypt the message both with her private key and Bob's public key (and of course also send the timestamp or the challenge). Doing so, an attacker will not be able to decrypt the message since he would need Bob's private key, and the communication can be secured.

Q5 Assume that the iptables firewall is running on host H, having a network interface eth1 (IP:192.168.0.1) connected to an internal LAN (IP: 192.168.0.0/24:the LAN is protected by H) and a network interface eth2 (IP: 151.100.5.5) connected to Internet. Assume that the default policy forall built-in chains is DROP. Then answer to Q5.1, Q5.2, Q5.3.

Q5.1 Define suitable rules to allow an HTTP server running on the machine 192.168.0.55 to correctly serve requests from the Internet.

> iptables -j ACCEPT -A FORWARD --dport 8080 -d 192.168.0.55 -i eth2

Q5.2 Explain the difference between the FORWARD,INPUT, and OUTPUT chains,presenting concrete examples (at least one for each chain) (you may consider at your choice host H, hosts from the internal LAN, and external hosts from the Internet).

The **forward** chain is that one where all packets are routed that are not for local delivery, so that they are just forwarded.
> iptables -j ACCEPT -A FORWARD -i eth2 -o eth1
The **input** chain is a list of rules that consider the incoming packets.
> iptables -j ACCEPT -A INPUT -i eth2
Analogously, the **output** chain deals with outgoing packages.
> iptables -j ACCEPT -A OUTPUT -o eth1

Q5.3 Define suitable rules for allowing users of the LAN to browse the web only in the case of HTTPS connections (HTTP traffic running via the standard port should be not allowed) using a standard browser.

> iptables -j ACCEPT -A FORWARD -s 192.168.0.0/24 --dport https_port -i eth1 -o eth2 -p tcp

Exam Refs

Go to: INDEX

1. Some exercises on math
2. Some exercises on symmetric cryptography
3. Some exercises on MAC and data integrity
4. Some exercises on Digital Signatures
5. Some exercises on authentication II
6. Some exercises on firewalls
7. Exam 2015 Jan 14th - Q1.1 : Describe what modes of operations are in the framework of block ciphers and the most relevant features to be considered for their analysis.
8. Exam 2015 Jan 14th - Q1.2 : Illustrate a mode of operations that makes the usage of a block cipher behaving similar to that of a stream cipher and discuss its security.
9. Exam 2015 Jan 14th - Q2.1 : Describe how RSA encryption/decryption work.
10. Exam 2015 Jan 14th - Q2.2 : Explain what is the mathematical relationship between e and d (the two exponents used in RSA).
11. Exam 2015 Jan 14th - Q2.3 : Why the textbook implementation of RSA is insecure? Provide at least an example.
12. Exam 2015 Jan 14th - Q5.1, Q5.2, Q5.3 : Some exercises from Mathematical Background II.
13. Exam 2015 Feb 10th - Q1 : Data Integrity of keyed HMAC and with AES.
14. Exam 2015 Feb 10th - Q2.1 : Diffie-Hellman vulnerabilities.
15. Exam 2015 Feb 10th - Q2.2 : Diffie-Hellman with three parties.
16. Exam 2015 Feb 10th - Q2 : Man In The Middle Attack
17. Exam 2015 Feb 10th - Q3.1 : Define the properties that qualify a hashing function as cryptographic.
18. Exam 2015 Feb 10th - Q3.2 : Describe the Merkle-Damgard construction. If the underlying hash function maps 256b blocks into 128b blocks, how many rounds are required for hashing a 140KB file?
19. Exam 2015 Feb 10th - Q3.3 : Possible schemes for keying a hash function
20. Exam 2015 Feb 10th - Q6.1, Q6.3 : Some exercises from Mathematical Background II.
21. Exam 2015 Feb 10th - Q6.2 : If p=13 and q=17, what is the range for exponent e?
22. Exam 2015 Jul 20th - Q2.1 : Describe in detail the Diffie-Hellman exchange of keys.
23. Exam 2015 Jul 20th - Q2.2, Q2.3 : Diffie Hellman and MITM attack.
24. Exam 2015 Jul 20th - Q5.4 : Define what a perfect cipher is.
25. Exam 2015 Jul 20th - Q5.5 : Explain what an adaptive chosen-plaintext attack is.
26. Exam 2015 Mar 26th - Q2.1 : Diffie Hellman Key Exchange.
27. Exam 2015 Mar 26th - Q2.2 : Diffie Hellman for three parties.
28. Exam 2015 Mar 26th - Q6.1, Q6.3 : Some exercises from Mathematical Background II.
29. Exam 2015 Mar 26th - Q6.2 : If p=13 and q=17, what is the range for exponent e?

30. Exam 2015 Nov 4th - Q5.2 : What is the Optimal Asymmetric Encryption Padding (OAEP) and why it provides “all or nothing” security?
31. Exam 2015 Nov 4th - Q5.3, Q5.4 : Some exercises from Mathematical Background II
32. Exam 2016 Feb 12th - Q1.1 : Describe the scenario of symmetric cryptography and define the concept of (synchronous and asynchronous) stream ciphers, block ciphers and modes of operations.
33. Exam 2016 Feb 12th - Q1.2 : Describe the RC4 cipher (both the key generation and the encryption process). What type of stream cipher is it?
34. Exam 2016 Feb 12th - Q1.3 : Describe and compare CBC and OFB. Can you suggest possible design criteria for their adoption?
35. Exam 2016 Feb 12th - Q6.1 : Exercise from Mathematical Background II
36. Exam 2017 Jul 24th - Q5.2 : Exercise from Mathematical Background II
37. Exam 2021 Jan 11th - Q1 : General concepts, true or false.
38. Exam 2021 Jan 11th - Q2.1 : Present in detail the main ideas behind RSA encryption/decryption.
39. Exam 2021 Jan 11th - Q2.2 : Discuss in detail at least two attacks against a textbook implementation of a RSA encryption/decryption.
40. Exam 2021 Jan 11th - Q2.3 : RSA and Digital Signatures.
41. Exam 2021 Jan 11th - Q3 : Cryptographic hash functions
42. Exam 2021 Jan 11th - Q4, Q5 : data integrity, data origin authentication, iptables