6CCS2CIS Cryptography and Information Security

Alex Franch Tapia

January 2019

Basic Knowledge

Computer security: deals with the prevention and detection of unauthorized actions by users of a computer system.

- The level of security is depended to the *security policy*, stating who or what performs which actions.

Network security: consists of the provisions made in an underlying computer network infrastructure, **policies** adopted by the **network administrator** to protect the network and the network-accessible resources from **unauthorized** access and the effectiveness (or lack) of these measures combined together.
Information security: is even more general: deals with securing information independent of computer systems.

- Knowing that someone is a user of a system is revealing of information which you may want to protect.

Formal definition*: protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction.*

Social engineering: is a way of attacking in which you exploit the human component of security systems.

Personae of Cryptography and Information Security

Agents are the different people or systems that interact in security.
Honest agents:

- Alice, Bob, Carol - agents communicating with each other like client and bank, bank and bank.

Dishonest agents:

- Eve and eavesdropper - i.e. a passive attacker that only listens.
- Charlie, Mallory and Zoe - malicious, active attackers.

Trusted and/or neutral agents:

- Simon and Trent - trusted servers
- Peggy and Victor - prover and verifier - zero-knowledge protocols

Two views on Information Security

- Security as policy compliance
- Security as risk minimisation

Security as policy compliance

Computer security deals with the prevention and detection of improper actions by users of a computer system. In security, a specification stipulates (in)acceptable system behaviors. E.g. Any user may fill a shopping cart but only authenticated users can go to check out.

For any system:

- Specification: What is it supposed to do?
- Implementation: How does it do it?
- Correctness: Does it really work?

In security:

- Specification: Policy
- Implementation: Mechanism (employed in system)
- Correctness: compliance

Security policy: states what system behavior is, and is not, allowed.
Security mechanisms: are used to enforce policy.

- Returning to analogy with correctness:

Formal Methods	Security
Specification ϕ	Security property ϕ
Program (or "Model") P	System P employing mechanisms
Correct: $P \models \phi$	Secure: $P \models \phi$

where $\models$ means "satisfies".

- Moreover, ϕ should hold for P in all malicious environments E, roughly $P||E \models \phi$ (i.e., P in parallel with E satisfies ϕ).

|| parallel lines tates that program P on top of environment E must satisfy the security property.

Traditional security properties/goals

Policies are drafter to achieve certain standard security properties (goals)
Common security properties are **CIA** (mnemonic)**:**

- **Confidentiality (secrecy):** no improper disclosure of information. Only those agents authorized are allowed to see the data.
- **Integrity:** no improper modification of information. Only authorized agents can change the data.
- **Availability:** no improper impairment of functionality/service. If a system should be running, it must be running.

What is (Im)proper must be specified for each system individually.

CIA deals with authorization (which requires some form of authentication and access controls). You have to prove that you are Luca Vigano to be able to authenticate him.

There are other definitions for CIA. Let's investigate.

Confidentiality

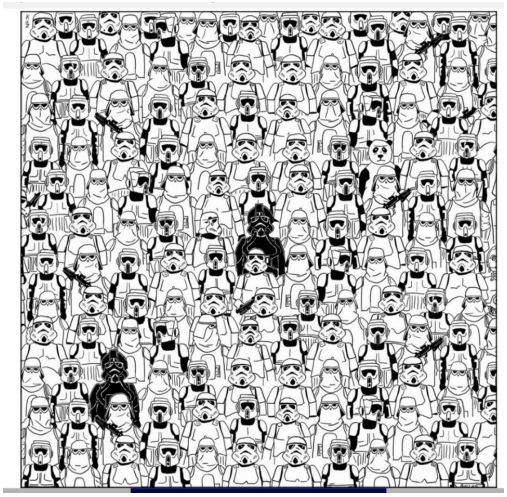
Confidentiality/secrecy: ensuring information is not seen by unauthorized principals.

- Passive attack on confidentiality: Alice and Bob are speaking and Eve listens. Eve is passive as it's not actually in the communication.
- An email is not a letter, rather a postcard. It can be read by everybody whilst it is in transit. If you want it to be confidential you have to implement a security mechanism be it network security, encryption and access control.
- **Privacy:** you choose what you let other people know. It deals with the confidentiality of information that you don't want to share.
- **Anonymity:** a condition where true identity is not known. Confidentiality of your identity (someone might be listening but they don't know who you are).

Confidentiality can go in many depths, there might be confidentiality of the message or of the communication and violating one might not mean violating the other.

Remember that the internet is designed as a public network, routing information is public (every IP package states its source and destination). Encryption does not hide identities - it hides payload but not routing information (even in IP-level encryption).

Anonymity



*The only mechanism we have to achieve anonymity we have to create an **anonymity set**.* We use a group that conveys multiple messages being the same of different to confuse. Alone you are exposed but the larger the set the more anonymous. When we build an anonymity service on the net, we need traffic. As seen in the image, there are many stormtroopers and a panda, it is difficult to find it. It states that the message doesn't have to be the same but similar.

If Alice send a message to Bob, she can encrypt it, but to be anonymous in front of Charlie they must send it within a set of messages.

We use Mix Networks on the net to create an anonymity set. To do so, we need huge traffic thus we need people to either send messages or dummy messages around the network. Some of these work but do come with a very high latency.

Attacks in Anonymity

Passive traffic analysis:

- Infer from network traffic who is talking to whom.
- To hide your traffic, must carry other people's traffic!

Active traffic analysis:

- Inject packets or put a timing signature on packet flow.

Compromise of network nodes (routers):

- It is not obvious which nodes have been compromised
    - Attacker may be passively logging traffic.
- Better not to trust any individual node
    - Assume that some fraction of nodes is good, don't know which.

Integrity

Integrity: data has not been (maliciously) altered.

- *Charlie will intercept communication between Alice and Bob, modify it and send it as if it were form Alice. This is an active attack. E.g. modifying a message to say pay to this bank account instead of this.*

Availability

Availability: data/services can be accessed when desired.

- *Alice wants to talk to Bob but Charlie cuts communication between the two.*

EXAM QUESTION: Define confidentiality, integrity, availability or accountability

Accountability

Accountability (non-repudiation): actions can be traced to responsible principals.

- *Alice cannot say she didn't send the message and Bob cannot say he did not receive it.*
- *In the case of digitally signing a contract, it is important to provide non-repudiation so that the person signing cannot say he never signed.*

***Non-repudiation:** actions done cannot be denied.*

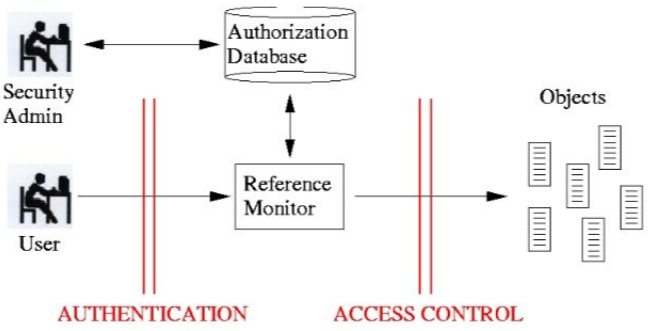
Authentication: principals or data can be identified accurately.

- *For all the above, it is necessary for people to be correctly authenticated.*

Methods for authentication are often:

- Something that you have - card
- Something that you know - password or secret key
- Something you are - a fingerprint, biometric

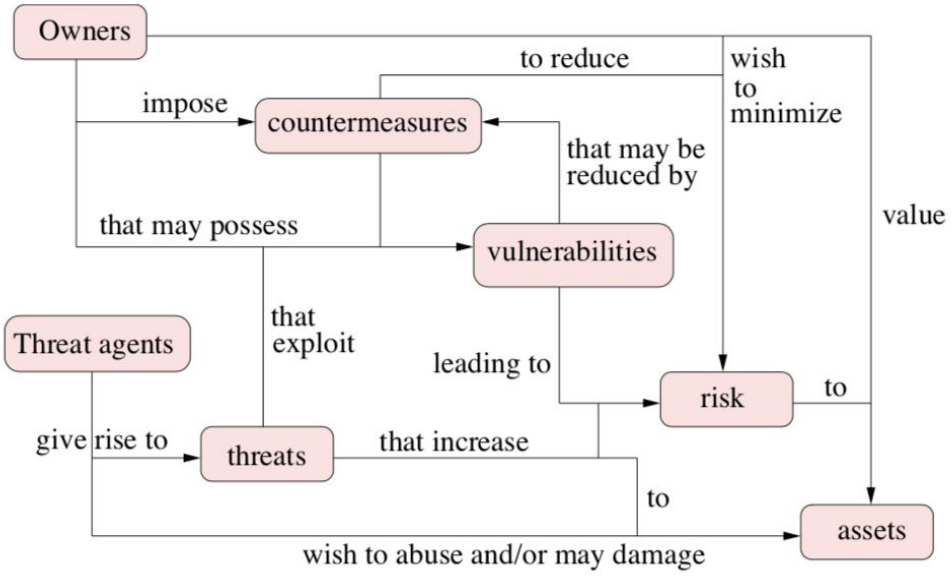
This is tightly bounded with authorization.



In summary, we test if a system is secure enough in the presence of an attacker.

Informational Security as risk minimization

Identifying owners, assets, threats, vulnerabilities and countermeasures.



We can quantify Risk by: risk = chance of abuse x impact.

Risk analysis and reduction steps

Part I. Analysis of existing risks:

1. Identify assets you wish to protect.
   What are the (information) assets and their functionalities?

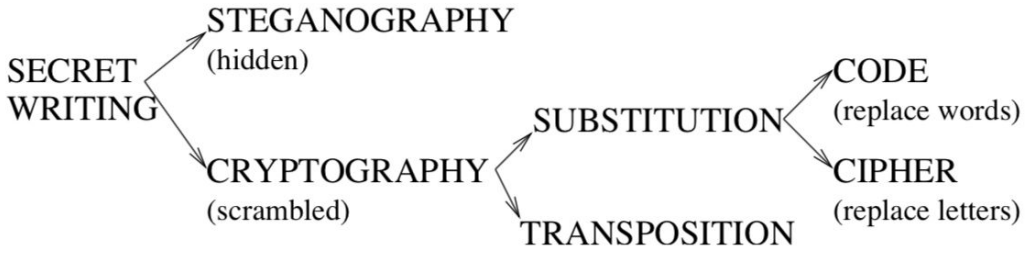
2. Identify risks to these assets.

Requires understanding threat agents and their threats as well as vulnerabilities.

Part II. Analysis of proposed security solution:

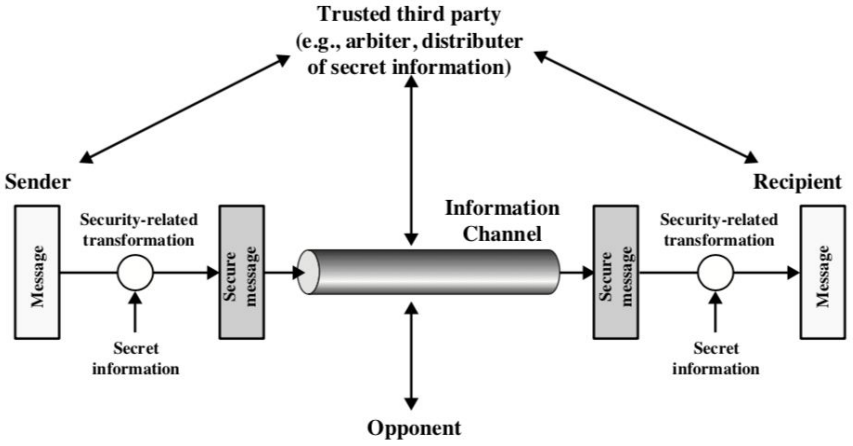
3. How well do proposed countermeasures reduce risk?
4. What other risks and tradeoffs do measures themselves bring?

Basic Concepts



Cryptography is the enabling technology to ensure that security properties are guaranteed.
Cryptology: the study of secret writing.
Steganography: the science of hiding methods in other messages.
Cryptography: the science of secret writing.
Cryptanalysis: science of recovering the plaintext from ciphertext without the key.

General model for network security



Information channel: a way to send information through the internet using a communication protocol (TCP/IP)

All security techniques have two components:

- A **security-related transformation** on the information sent. For example the **encryption** of the message or the addition of a **code** based on the contents of the message to identify a sender (MAC)
- Some **secret** information shared by two principals and hopefully unknown to the opponent. For example an **encryption key.**

Sometimes we might need:

- A trusted third party, to achieve secure transmission. For example: someone that distributed secret information to two principals while keeping it from an opponent or someone that resolves disputes between two principles concerning the authenticity of a message.

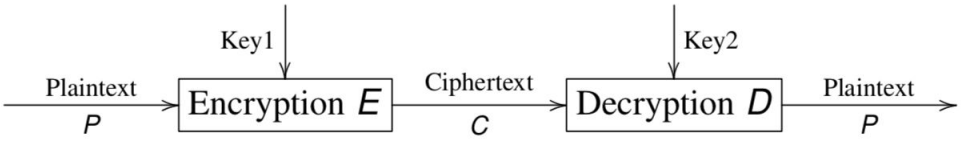
In the general model there are **4 different basic tasks**:

- Design an algorithm that performs transformations. An opponent shouldn't be able to defeat the purpose of this.
- Generate secret information to be used with the algorithm.
- Develop methods for the distribution and sharing of the secret information

- Specify a protocol to be used that makes use of the security algorithm and information to achieve a particular security service.

You cannot encrypt something if you have not decided on something before.

General cryptographic schema



where $E(\text{Key1}, P) = C$ and $D(\text{Key2}, C) = P$.

P: Plaintext (plain text, clear text, ...): text that can be read and “understood” (e.g., by a human being).
E: Encryption: transformation (function, process, procedure, ...) E that takes in input a plaintext and a key and generates a ciphertext.
C: Ciphertext (cipher text, encrypted text, ...): transformed (scrambled, ...) text that needs to be “processed” to be “understood” (e.g., by a human being).
D: Decryption: transformation (function, process, procedure, ...) D that takes in input a ciphertext and a key and generates a plaintext.
Cipher: a function (algorithm, ...) for performing encryption/decryption.
Symmetric algorithm: key1 = key2, or are easily derived from each other. If you know one, you know the other.
Asymmetric (or public key) algorithms: use different keys, which cannot be derived from one another. There are two keys, public and private.
Generally:

- *Encryption and decryption should be easy, if keys are known.*
- *Security should depend on the secrecy of the key, not on the algorithm.*
    - *Often times algorithms are public and known so that they can be implemented in hardware for efficiency purposes.*

General Encryption and Decryption Formula

Encryption:
C = E(P,K)
Decryption:
P = D(C,K)

Note: do not forget that an **encryption function** and a **cipher** are different. We will encrypt using an encryption function which is part of an **encryption scheme/cipher.**

A Mathematical Formalization

- $\mathcal{A}$, the alphabet, is a finite set.
- $\mathcal{M} \subseteq \mathcal{A}^*$ is the message space. $M \in \mathcal{M}$ is a plaintext (message).
- $\mathcal{C}$ is the ciphertext space, whose alphabet may differ from $\mathcal{M}$.
- $\mathcal{K}$ denotes the key space of keys.
- Each $e \in \mathcal{K}$ determines a bijective function from $\mathcal{M}$ to $\mathcal{C}$, denoted by E_e. E_e is the encryption function (or transformation).

  Note: we will write $E_e(P) = C$ or, equivalently, $E(e, P) = C$.
- For each $d \in \mathcal{K}$, D_d denotes a bijection from $\mathcal{C}$ to $\mathcal{M}$. D_d is the decryption function.
- Applying E_e (or D_d) is called encryption (or decryption).

Examples:
A = {a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,w,x,y,z}
A* = alex
Message space: the set of possible messages
Ciphertext space: the set of all possible ciphertexts
Key space: the set of all possible keys
Message: jus like we do in english some sequences of characters have meaning and others don't but they are all inside the message space.

Mnemonic to remember: **bAMmCKed** -> which sounds like BUM, BACK and HEAD put together. Bijections, Alphabet, Message Spacem, message, Ciphertext space, Key space, encryption key and decryption key in order.

Bijective function: a function in which you can go and come back from input to output with a one-to-one correspondence. Example: given a function f : a -> b, meaning f(a) = b, we can go from a to b but we can also do the opposite go from b to a, f(b) = a, not only that but a will always be mapped to be and viceversa and only mapped to that one other output.

This is important to make sure the plaintext created through an encryption function will be the same as the one when we decrypt the ciphertext generated. Otherwise after encryption, decryption would lead to a different plaintext.

Exam: the mathematical formalisation is asked many times, being able to reproduce it is important. Question about why it is important that functions are bijective is also very present.

A more formal explanation of the bijection follows:

- An encryption scheme (or cipher) consists of a set $\{E_e \mid e \in \mathcal{K}\}$ and a corresponding set $\{D_d \mid d \in \mathcal{K}\}$ with the property that for each $e \in \mathcal{K}$ there is a unique $d \in \mathcal{K}$ such that $D_d = E_e^{-1}$; i.e.,

$$D_d(E_e(m)) = m \qquad \text{for all } m \in \mathcal{M}\,.$$

- The keys e and d above form a key pair, sometimes denoted by (e, d). They can be identical (i.e., the symmetric key).

- To construct an encryption scheme requires fixing a message space $\mathcal{M}$, a ciphertext space $\mathcal{C}$, and a key space $\mathcal{K}$, as well as encryption transformations $\{E_e \mid e \in \mathcal{K}\}$ and corresponding decryption transformations $\{D_d \mid d \in \mathcal{K}\}$.

Encryption Scheme definition for symmetric encryption

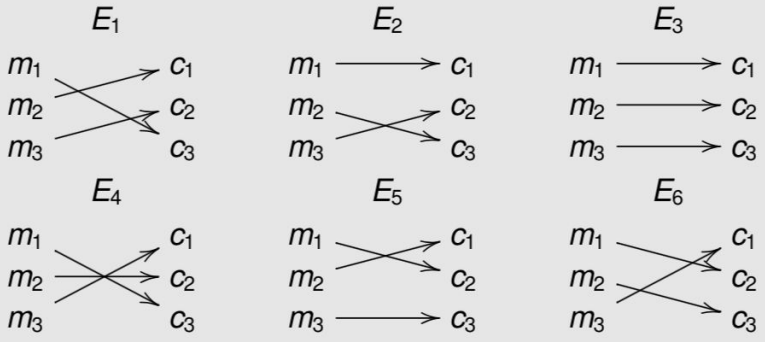
An encryption scheme with $\{E_e \mid e \in \mathcal{K}\}$ and $\{D_d \mid d \in \mathcal{K}\}$ is symmetric-key if for each associated pair (e, d) it is computationally "easy" to determine d knowing only e and to determine e from d. In practice $e = d$.

Encryption Scheme definition for symmetric encryption

An encryption scheme with $\{E_e \mid e \in \mathcal{K}\}$ and $\{D_d \mid d \in \mathcal{K}\}$ is asymmetric-key if knowing E_e it is infeasible, given $c \in \mathcal{C}$, to find an $m \in \mathcal{M}$ such that $E_e(m) = c$. This implies that it is infeasible to determine d (the private key) from e (which is called public key as it can be public information). Hence, E_e constitutes a trap-door one-way function with trapdoor d.

An example

Let $\mathcal{M} = \{m_1, m_2, m_3\}$ and $\mathcal{C} = \{c_1, c_2, c_3\}$.
There are 3! = 6 bijections from $\mathcal{M}$ to $\mathcal{C}$.
The key space $\mathcal{K} = \{1, 2, 3, 4, 5, 6\}$ specifies these transformations.



Suppose Alice and Bob agree on the transformation E_1.
To encrypt m_1, Alice computes $E_1(m_1) = c_3$.
Bob decrypts c_3 by reversing the arrows on the diagram for E_1 and observing that c_3 points to m_1.

We can calculate the number of bijections by looking at the cardinality (number of elements) of the message space factorial. If there are 3 elements in the message space then **3! = 6**

(# e in MessageSpace)! = # bijections

The number of bijections also tell us about the number of keys that we need (given that encryption and decryption keys are the same).

Symmetric Encryption

- An encryption scheme $\{E_e \mid e \in \mathcal{K}\}$ and $\{D_d \mid d \in \mathcal{K}\}$ is symmetric-key if for each associated pair (e, d) it is computationally "easy" to determine d knowing only e and to determine e from d. In practice $e = d$.
  - Also known as: secret-key, single-key, one-key, shared-key, conventional encryption.
  - Sender and recipient share a common key.
  - All classical encryption algorithms are symmetric-key (it was only type of encryption prior to invention of public-key crypto in 1970's).
  - Most widely used.

Requirements for Secure Use of Symmetric Encryption

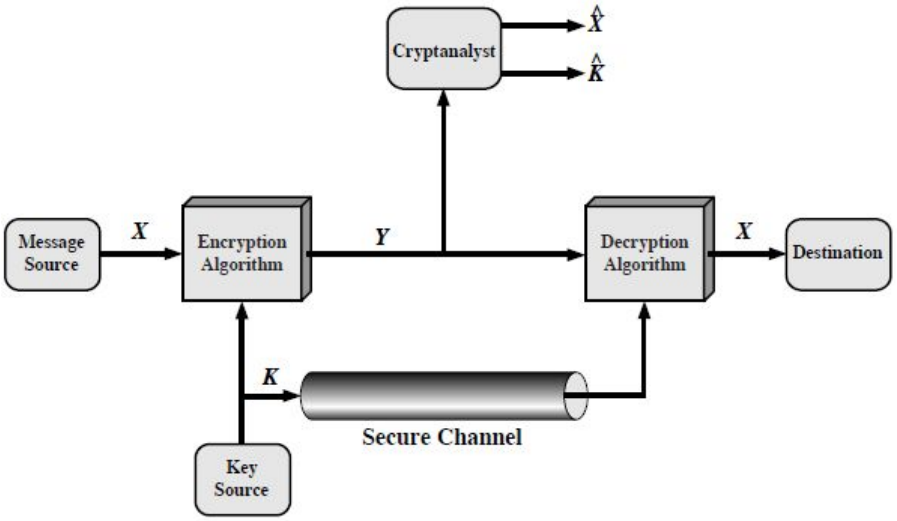
1. **A strong encryption algorithm**
   a. **At a minimum:** an attacker that has access to one or more ciphertexts shouldn't be able to decipher ciphertext or figure out the key.

   b. **Stronger:** an attacker shouldn't be able to decrypt ciphertext or discover the key even if he has many ciphertexts with their plaintexts.
2. **Key exchange must be done in a secure way and must be stored securely** (e.g. exchanging them across a secure channel)

For a good cryptographic algorithm we don't keep the algorithm secret we only keep the **key secret.**

- We assume that it is impractical to decrypt given a ciphertext plus knowledge of encryption/decryption algorithm.
- This makes symmetric encryption feasible for widespread use since:
    - Manufacturers can have encryption algorithms implemented on low-cost chips
    - Chips are widely available and can be incorporated to many products.

Detailed Model of Symmetric Cryptosystem



- The message X is composed of elements of some finite alphabet. Nowadays we typically use A = {0,1}
- A key K is composed of elements in the key space and must be transmitted in a secure way either from the message source or distributed by a third party that delivers it to both parties.
- Encryption algo outputs a ciphertext (Y in the diagram) Y = E(K , X)
- The intended receiver with the Key can reverse encryption via decryption and generate the plaintext again. X = D(Y, K)

Attacker:

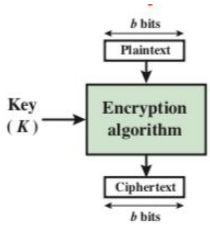
- Knows the encryption algorithm **E** and the decryption algorithm **D**.
- It can observe **Y** but doesn't have access to K or X and he may attempt to recover these by generating ^K or ^X.

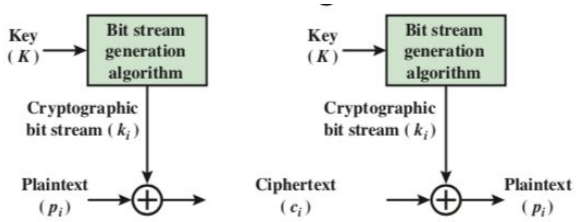
Characteristics of Cryptographic Systems

1. Type of operations used to transform plaintext into ciphertext.
    - **Substitution:** each element (bit, bits, letter or group of letters) in plaintext is mapped into another element.
    - **Transposition:** elements in plaintext are rearranged.

   *All **operations** must be **reversible**, else information is lost.*
2. Number of keys used
    - **Symmetric,** one key or **asymmetric** different keys
3. How is the plaintext processed or *how big are your elements.*
   Are you going to process everything via single bits or chunks.
    - **Block cipher** processes input one block of elements at a time, producing an output block for each input block.

- **Stream cipher** processes input elements continuously, producing in output one element at a time, as it goes along. Ie processing one bit at a time. You take one element of the plaintext and encrypt it with one element of the key and then you get one element of the ciphertext.





We can also use **codes**. Codes are a way of simplifying plaintext by mapping our code to the plaintext. Ie we might map “The” to 1701 or “secret” to 2831 and then write 1701 2831 to say The Secret.

- **Problem:** If a word we are trying to process is not in the **code-book** (where the table of mappings is) then you cannot encode it.

Cryptanalysis and brute-force attacks

The aim of attacking an encryption system is to recover the **key** in use so that we can decrypt all past and future messages. The security of a system is typically attacked in a systematic way. There are two ways to attack an encryption scheme:

Cryptanalysis: attacks rely on knowing the algorithm plus some knowledge of the plaintext or even some plaintext-ciphertext pairs. It exploits the characteristics of the algorithm to attempt to deduce a specific plaintext or to deduce the key being used.

Brute-force attack: attacker tries every possible key on a piece of ciphertext until an intelligible translation into plaintext is obtained. The cost heavily depends on the key size. ***On average, half of all possible keys must be tried to achieve success.***

- Each element in a key can be a { 0, 1 } thus number of keys are 2^(length of key)
- This attack is always possible, keep trying decryption with different keys until you get an intelligible plaintext (assuming we know the plaintext)

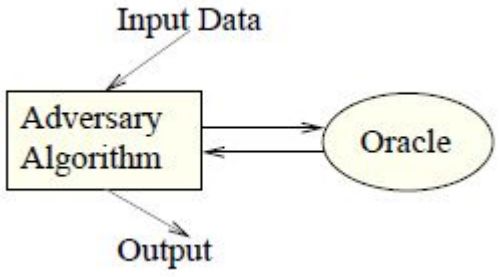
General formula for number of keys: **#keys = (#elements in key space) ^ length of key**

Security by obscurity: *we don’t tell them which alphabet and which algorithm we are using. In Cryptography we assume that the alphabet and the algorithm is known.*

Instead we publish the algorithms so that we quickly test how secure you are so many people have tried to break them.

Model of Attack

Given some input, will a malicious algorithm be able to produce the correct output? We can think of an adversary playing a game:



Input: whatever an adversary know at the beginning (e.g. public key, plaintexts…)

Oracle: models/simulates the information necessary an attacker can obtain during the attack. (resources available given the type of attack)

Output: what the attacker is looking to compute (secret key, partial information of plain text …)

He wins if he succeeds.

Types of Attack

- **Ciphertext only:** cryptanalyst knows the encryption algorithm and ciphertext.
- **Known plaintext:** cryptanalyst know the encryption algorithm, ciphertext and one or more ciphertext-plaintext pairs generated with the key.
- **Chosen plaintext:** cryptanalyst knows the encryption algorithm, ciphertext and a plaintext chosen by the cryptanalyst with its ciphertext pair generated with the secret key.
- **Chosen ciphertext:** knows the encryption algorithm, ciphertext, ciphertext chosen by analyst along with its corresponding plaintext generated with the secret key.
- **Chosen text:** encryption algorithm, ciphertext, plaintext chosen by analyst along with corresponding ciphertext and ciphertext chosen by cryptanalyst with the decrypted plaintext under the secret key.
- **Adaptive chosen plaintext:** cryptanalyst can choose plaintext and he can modify it based on encryption results.

To define security we have to:

1. Specify an oracle (or type of attack)
2. Define what the adversary required to win the game.
3. Conclude if the system is secure under the definition, if any efficient adversary wins the game with only negligible probability.

Standard Security Definition for Conventional Encryption

- No input data for adversary
- Choose plaintext attack:
    - Case 0: we encrypt a message m and the oracle returns m encrypted under a **fixed key** randomly chosen; or
    - Case 1: oracle returns encryption of a randomly chosen message, independent of m.

In case 1 the attacker gets useless data, if he cannot tell the difference apart from correct encryptions from random ones then he cannot do damage in the real world. In other words: if the cryptanalyst cannot be certain if the bits of the random message are the ones from the original message or not then he cannot establish a connection between the ciphertext and its plaintext, thus he can do no harm.

Substitution techniques

A ***substitution technique*** is one in which the letters of plaintext are replaced by other letters, numbers or symbols.

KHOOR ZRUOG = **HELLO WORLD**
Caesar cipher: each plaintext character is replaced by character 3 to the right modulo 26.

Cesar Cipher

Earliest and simplest substitution cipher. Mathematically we give each letter a number (0 to 25):

a	b	c	d	e	f	g	h	i	j	k	l	m
0	1	2	3	4	5	6	7	8	9	10	11	12

n	o	p	q	r	s	t	u	v	w	x	y	z
13	14	15	16	17	18	19	20	21	22	23	24	25

We can use any key we want. But 0 would be a pretty stupid key.

then: $C = E(3, P) = (P + 3) \mod 26$

In general, for $K \in \{1, \ldots, 25\}$:

$$C = E(K, P) = (P + K) \mod 26$$
$$P = D(K, C) = (C - K) \mod 26$$

True force attack is very easy because:

- The key space is very small - only 25 keys to try
- We know the **encryption** and **decryption** algorithm.
- The language of the plaintext is known and recognizable

Mnemonic of how easy a brute-force attack is: KLED. Low number of Keys to try, the Language is known and the Encryption and Decryption algorithms are known.

Exam Q: Using multiplication for Cesar Cipher?

Multiplication modulo n is not a bijective function, it is not reversible. Given a key 2 and A = 0, B = 1,.. N = 13, then both will be encrypted to 0. Since 0 mod 26 = 0 and 26 mod 26 = 0, then we will not know how to get back.

Think of division and exponentiation

Monoalphabetic substitution Ciphers

Mono or single alphabetic means we are only using a single alphabet. We achieve a monoalphabetic substitution cipher

Possible permutations of a set of n elements: n!
(1st element can be chosen in 1 of n ways, 2nd in n − 1 ways, etc.)

In this case we have a mapping of each letter to another but neither the inner or alphabet is ordered, therefore an arbitrary letter could map to any of the 26 characters.

A cesar cipher is an example of a monoalphabetic substitution cipher in which the alphabets are ordered and there is a shift of 3 letters.

Let $\mathcal{K}$ be the set of all permutations on the alphabet $\mathcal{A}$.
Define for each $e \in \mathcal{K}$ an encryption transformation E_e on strings $m = m_1 m_2 \cdots m_n \in \mathcal{M}$ as

$$E_e(m) = e(m_1)e(m_2)\cdots e(m_n) = c_1 c_2 \cdots c_n = c$$

To decrypt c, compute the inverse permutation $d = e^{-1}$ and

$$D_d(c) = d(c_1)d(c_2)\cdots d(c_n) = m$$

E_e is a **mono-alphabetic substitution cipher**.

Given that any letter could be matched to any other and the alphabet is ordered, there would be 26! Keys, these are a lot of keys. Meaning it would be difficult to crack it using a brute force attack but it is easy to break it by using **frequency analysis**.

Frequency Analysis

Is a technique that exploits the constructs of a language. We look at the relative letter frequencies and can determine which letter they are, simply by looking at the relative letter frequencies in that language (ie English language).

If in a ciphertext sufficiently large P and Z occur the most, they are likely to correspond to E and T, since these are the most used letters in the english alphabet. We can also look at **digram** (sequences of 2 letters) frequencies.

Homophonic Substitution Ciphers

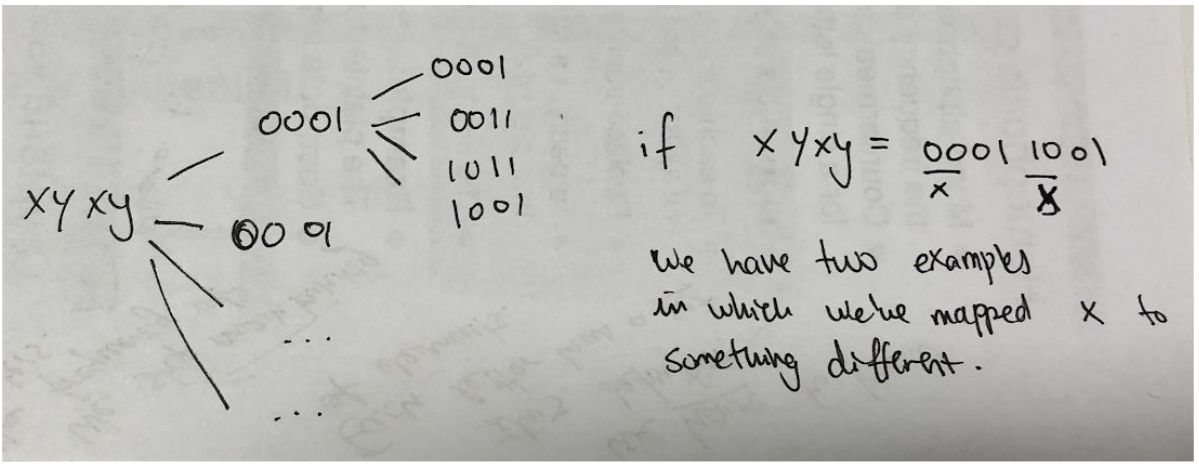
Provide multiple substitutes for a single letter. Mapping a single plaintext character to a set of values.

Homophonic substitution cipher

- To each $a \in \mathcal{A}$ associate a set $H(a)$ of strings of t symbols, where $H(a), a \in \mathcal{A}$ are pairwise disjoint.
- Replace each a with a randomly chosen string from $H(a)$.
- To decrypt a string c of t symbols, one must determine an $a \in \mathcal{A}$ such that $c \in H(a)$.
- The key for the cipher is the sets $H(a)$.

- **Example:** $\mathcal{A} = \{x, y\}$, $H(x) = \{00, 10\}$, and $H(y) = \{01, 11\}$.
  The plaintext xy encrypts to one of 0001, 0011, 1001, 1011.

- We are making encryption more difficult and therefore decryption.
- We are making cryptanalysis more complicated but still possible because letter frequencies are still there just not that obvious.
  - Each element of the plaintext only affects each element of the ciphertext
  - Multiple letter patterns still survive in the ciphertext



There are two methods we can use to lessen the survival of the structure of plaintext in the ciphertext.

- Encrypt multiple letters of plaintext - **Playfair Cipher**
- Use multiple cipher alphabets (polyalphabetic substitution) - **Vigenere Cipher**

Playfair Cipher

Encrypts multiple letters of the plaintext at the same time.

Note, it must always use the same filler letter, if not we start creating words.

Playfair Cipher 1

PLAINTEXT : BALLOON

KEY : MONARCHY

1. Split plaintext into groups of 2 characters.

   BALLOON → BA LL OO N

   - If any two letters repeat in a single group, add a filler letter (e.g. X) to break that.

     BA LL OO N → BA LX LO ON

     NOTE: We no longer have two O's together.

   - If any group is not complete add a letter to fill it.

     NOT THE CASE. BALXLOON has 6 letters.

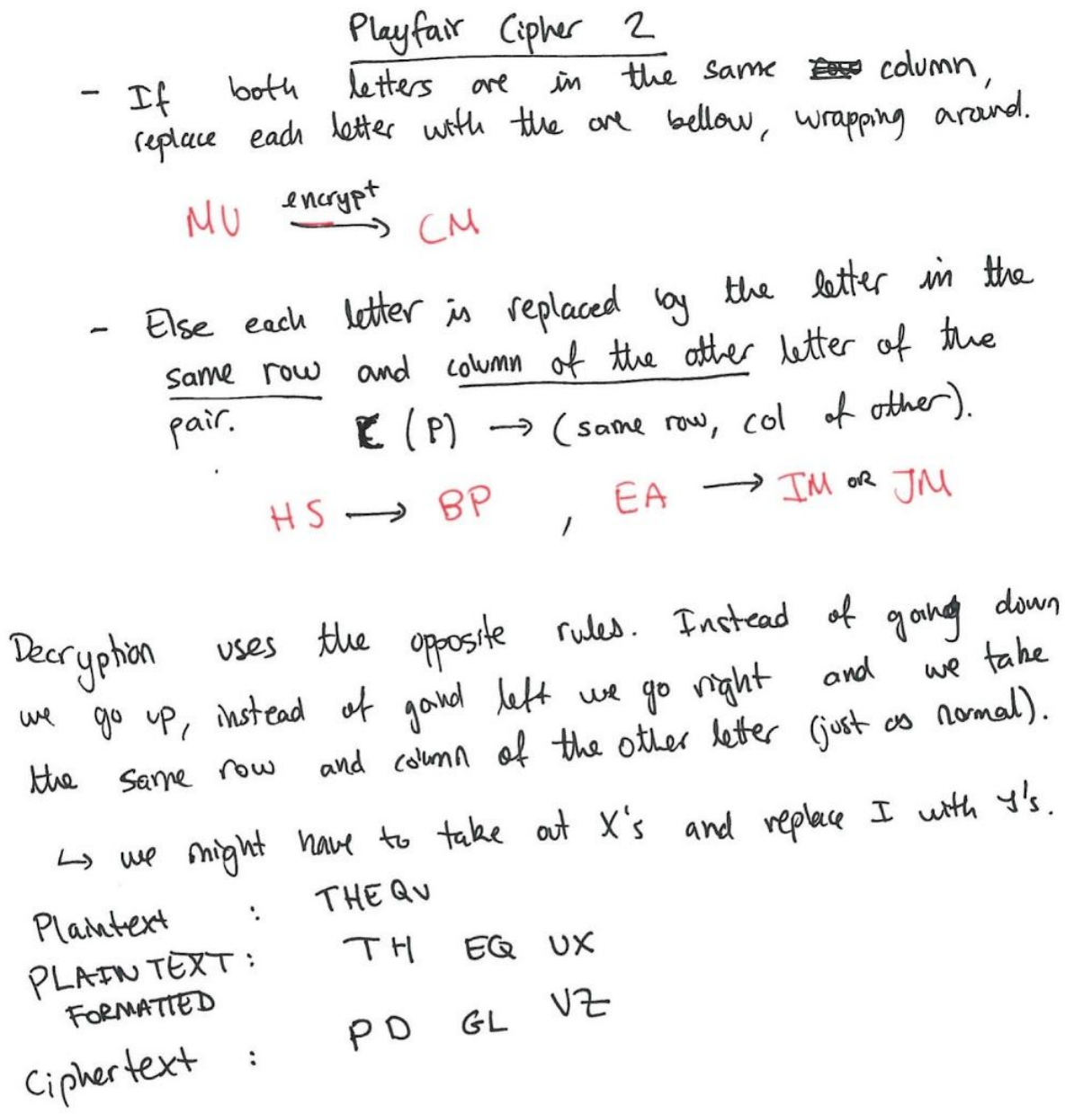
2. Build a matrix 5×5 with the keyword at the top, then fill it with the rest of the alphabet. (omit letters that appear twice in our key and fill up with letters not in our keyword).

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

3. Encrypt our plaintext.

   - If the two letters in the pair are in the same row. Replace letter with letter to the right, wrapping around.

     AR —encrypt→ RM



Playfair Security

Much better than monoalphabetic since we are using diagrams and each letter has 26 possibilities.

- 26 x 26 = 676 digrams vs 26 letters

However breaking it is relatively easy:

- Since it leaves much of the structure of the plaintext languages, there are some X's but digrams keep the structure.
- A few hundred letters of ciphertext is enough to reconstruct the keyword and the plaintext

Vigenere Cipher

It's a polyalphabetic substitution cipher. It changes the alphabet everytime we substitute an element. Essentially it means,

Take my monoalphabetic substitution cipher, encrypt a letter, turn the wheel and encrypt another word. We are talking another alphabet.

You have to agree how to change the alphabet.

Vigenere cipher: a set of monoalphabetic substitution rules which consists of the 26 cesar ciphers with shifts of 0 - 25. The key is denoted by the letter that substitutes **A**. A CC with a shift of 3 has a key value of **D**.

VIGENÈRE CIPHER

We have:

- A plaintext $P =$ WE ARE DISCOVERED SAVE YOURSELF

  $P = p_0, p_1, p_2, \cdots, p_{n-1}$

  W E A F

- A key $K =$ DECEPTIVE

  $K = k_0, k_1, \cdots, k_{m-1}$

  D E E

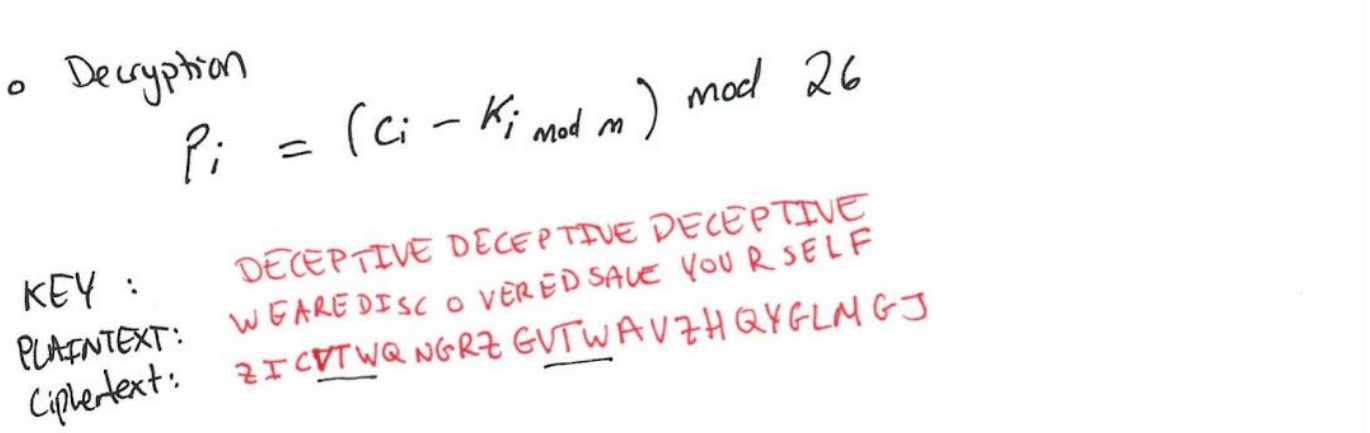
Normally the key is shorter than P. $m < n$

- A ciphertext. C → Encryption
  - ↳ To find C, we encrypt every single element of P with every element of K, repeating the key if necessary.

$C = c_0, c_1, \cdots c_{n-1}$

$= E(K, P)$

$C_i = (p_i + K_i \bmod m) \bmod 26$



The vigenere cipher allowed for a breakthrough in cryptography, showing that we need keys which are more complex than just numbers.
There are multiple ciphertext letters for each plaintext letter, frequency information is better than playfair, it is obscured but considerable frequency info remains (as seen in underlined example).

To break it: we find two or more collisions a specific distance apart, then you can estimate the length of the key and attack each monoalphabetic cipher.

Vernam Cipher: XOR

Just like an or but ONLY true if we have **one positive and one negative**.

It works on binary data (bits) rather than letters, using **XOR** $\oplus$:

$$\begin{aligned} 0 \oplus 0 &= 0 \\ 0 \oplus 1 &= 1 \\ 1 \oplus 0 &= 1 \\ 1 \oplus 1 &= 0 \end{aligned}$$

so that

$$\begin{aligned} a \oplus a &= 0 \\ a \oplus 0 &= a \\ a \oplus b &= b \oplus a \\ a \oplus b \oplus b &= a \\ (a \oplus b) \oplus c &= a \oplus (b \oplus c) \end{aligned}$$

XOR can be used as polyalphabetic cipher:

$$\begin{aligned} P \oplus K &= C \\ C \oplus K &= P \end{aligned}$$

XOR is essentially addition modulo 2.

We perform a bit by bit XOR between the key and ciphertext.

+ Difficult to break if the key is long (at least as long as the ciphertext and without repetition)
- Still breakable with sufficient ciphertext

Example of how it achieves perfect secrecy

Given a business man that wants to make the decision of buying (increase company shares) or sell (decrease company shares) and that this person is going to transmit this to his broker over the public network. We have a 50% chance of guessing which. Mr X decides to encrypt his message, since a substitution cipher could be so easy to break given the short word, we decide to use a system with two keys, K1 and K2 which are equally likely:

- K_1 encrypts "buy" to 0 and "sell" to 1:
  $$E_{K_1}(\text{"buy"}) = 0 \text{ and } E_{K_1}(\text{"sell"}) = 1.$$
- K_2 encrypts "buy" to 1 and "sell" to 0:
  $$E_{K_2}(\text{"buy"}) = 1 \text{ and } E_{K_2}(\text{"sell"}) = 0.$$

The attacker has no way to know what 1 or 0 means if he intercepts the message because both keys are equally likely. Of course MR X and the Broker must have met before and explained what each key means.

- Before we intercept ciphertext, we can only guess.
- Once we intercept ciphertext, the attacker can guess the key
- Since the #keys = #messages, the chances of either guess being correct are equal.

This is **perfect secrecy**.

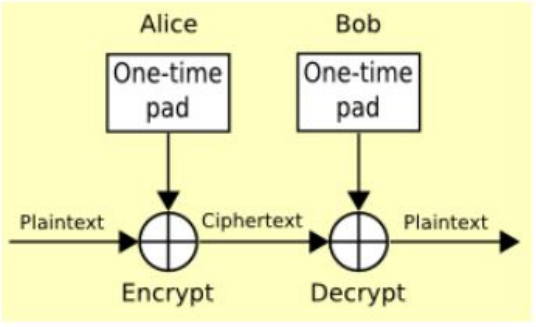
One-time pad

Uses a truly random key that is:

- As long as the message, so the key is not repeated.
- Use it to encrypt and decrypt a single message, then we discard it.

Every time we have a new message we need a new key of the same length as the message. This produces a random output with no statistical relation to plaintext.

- **Unbreakable:** C contains no information whatsoever about P.
- Only cryptosystem that exhibits perfect secrecy.
- We must have a booklet, with a bunch of random keys that we have decided to use for us to use the same key when encrypting and decrypting and then crossing it out to never use it again.



Given that the keys were produced in a true random fashion and that they are used only once, then we know that there are not patterns or regularities in our ciphertext.

However:

- Producing truly random large random keys is difficult
- You must agree which key you are using (as an attacker I will try to attack this, he will try to get the booklet)
- The key must be of equal length for both parties.

Normally used for low-bandwidth channel requiring very high security.

Transposition Ciphers

Build anagrams, sentences in which the order of the letters has been changed. It works on blocks of letters of the plaintext.

Encryption: the shift (or permutation) of the letters of the plaintext.
Decryption: the inverse shift (inverse permutation) of the letters of the ciphertext.

- Letters unchanged so one can exploit frequency analysis.

Rail Fence Cipher

Plaintext is written down as a sequence of diagonals and then read off as a sequence of rows.

For example, to encipher the message

MEET ME AFTER THE TOGA PARTY

with a rail fence of depth 2, we write:

M E M A T R H T G P R Y
E T E F E T E O A A T

so that the ciphertext is

MEMATRHTGPRYETEFETEOAAT

Note that we can have more depths.
This is trivial to cryptanalyse.

Note: there is something called the zig-zag cipher which is similar but uses a zig zag to represent text.

with a “standard” rail fence of depth 3, we write:

W R I O R F E O E
E E S V E L A N
A D C E D E T C

so that the ciphertext is

WRIORFEOEEESVELANADCEDETC

whereas with a zig zag cipher of depth 3, we write:

W E C R L T E
E R D S O E E F E A O C
A I V D E N

so that the ciphertext is

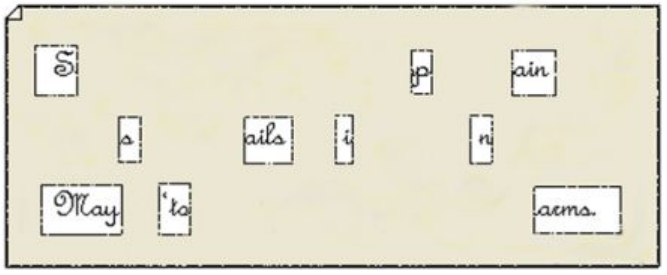
WECRLTEERDSOEEFEAOCAIVDEN

Very easy, we are looking for something more complex

Rotating (Turning) Grilles

You write text in a meaningful way but some letters in it belong to the plaintext and some don’t.

Sir John regards you well and spekes again that
all as rightly 'vails him is yours now and ever.
May he 'tone for past a'lays with many charms.



We need the same grid to encrypt as to decrypt.
This is an example of steganography

Rotating turning grille: the same idea but we cut out 9 squares out of a 6x6 grid and write your ciphertext there, then you rotate it 90 deg and write it in the squares that are now visible, and so on and so forth.

Multi-stage Columnar Transposition Cipher

We write a message in a rectangle, row by row and read the message off column by column, but we permute the order of columns.

For example, with key 4312567 (and with padding to fill the grid)

Key:	4	3	1	2	5	6	7
Plaintext:	a	t	t	a	c	k	p
	o	s	t	p	o	n	e
	d	u	n	t	i	l	t
	w	o	a	m	x	y	z
Ciphertext:	TTNAAPTMTSUOAODWCOIXKNLYPETZ						

This is easily recognised and attacked. The ciphertext has the same letter frequencies.
We can also do a **multiple-stage columnar transposition cipher**. Which is performing the same more than once, which increases the security. We can visualise the increase of the security:

First transposition, still quite regular structure (+7 in blocks of 4!):

03	10	17	24	04	11	18	25	02	09	16	23	01	08
15	22	05	12	19	26	06	13	20	27	07	14	21	28

Second: less structured permutation (cryptanalysis more difficult):

17	09	05	27	24	16	12	07	10	02	22	20	03	25
15	13	04	23	19	14	11	01	26	21	18	08	06	28

Steganography

Conceal the existence of a message within a larger message.

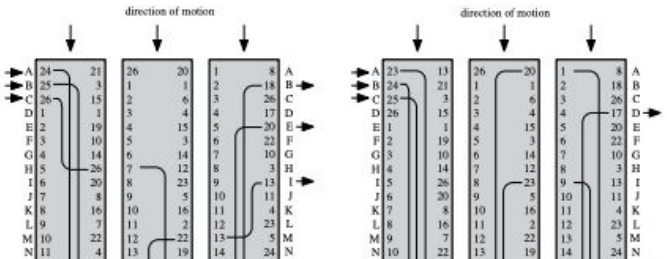
- Arrangement of words or letters (the first letter of each word creates another message)
- Invisible ink
- Pin punctures on paper
- \- Requires a lot of work to hide a small amount of info
- \- Once the system is discovered it's worthless
- \+ Done by parties who have something to hide

Composite (product) Ciphers

Product ciphers chain substitution-transposition combinations to create something that is more secure.

- 2 substitutions, are really one one more complex substitution and 2 transpositions are really only one transposition but **substitution** followed by **transposition** makes a new harder cipher.

Historically, in WW2, rotors were used. We could use 3 cylinders in which each cylinder gives you one substitution, then you would rotate each cylinder.
With 3 cylinders we have $26^3 = 17576$ alphabets.



You start encrypting until you've done one full turn of the left-most cylinder, then turn one in the middle cylinder and so on and so forth (a bit how we would break a lock)

We substitute with each rotor but the combination of them are transpositions.

Feistel Cipher

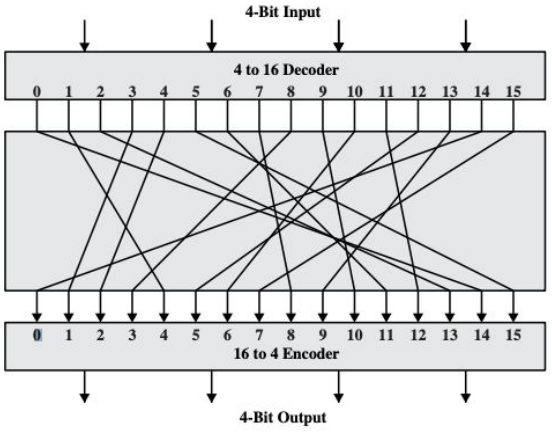
Nearly always in the exam.

Motivation for the Feistel Cipher

If we have a block cipher that encrypt n-bits into n-bit ciphertext.
Input: a plaintext of n bits or splits it into n bits
Output: ciphertext of n bits or blocks of n bits

- There are $2^n!$ different transformations because there are 2^n possible plaintext blocks (each bit can have two values and there are n bits) and we need reversible mappings, thus for the first option I have 2^n options, for the second $2^n - 1$, …

We must make sure that every input is mapped **exactly** to one output, to ensure it is reversible.



Problems:

- **Small block size**: it is equivalent to a substitution cipher and easily attackable
- **Large block size:** these would generate really large keys and would make the implementation difficult and wouldn't perform very well.

Feistel suggested: invertible product cipher. Which is an approximation to ideal block cipher for large n, built out of components which are easily realizable.

Feistel Cipher

Execution of 2 or more simple ciphers in sequence to that the product is cyptographically stronger than any component.

- Idea: cipher with k-bit key and n-bit blocks, allows a total of 2^k possible transformations rather than $2^n!$.
- Alternates substitutions and permutations (transpositions)

S-boxes and P-boxes

The feistel cipher is a practical application of confusion and diffusion functions.
Product ciphers chain combinations of substitutions and transpositions/permutations such that the:

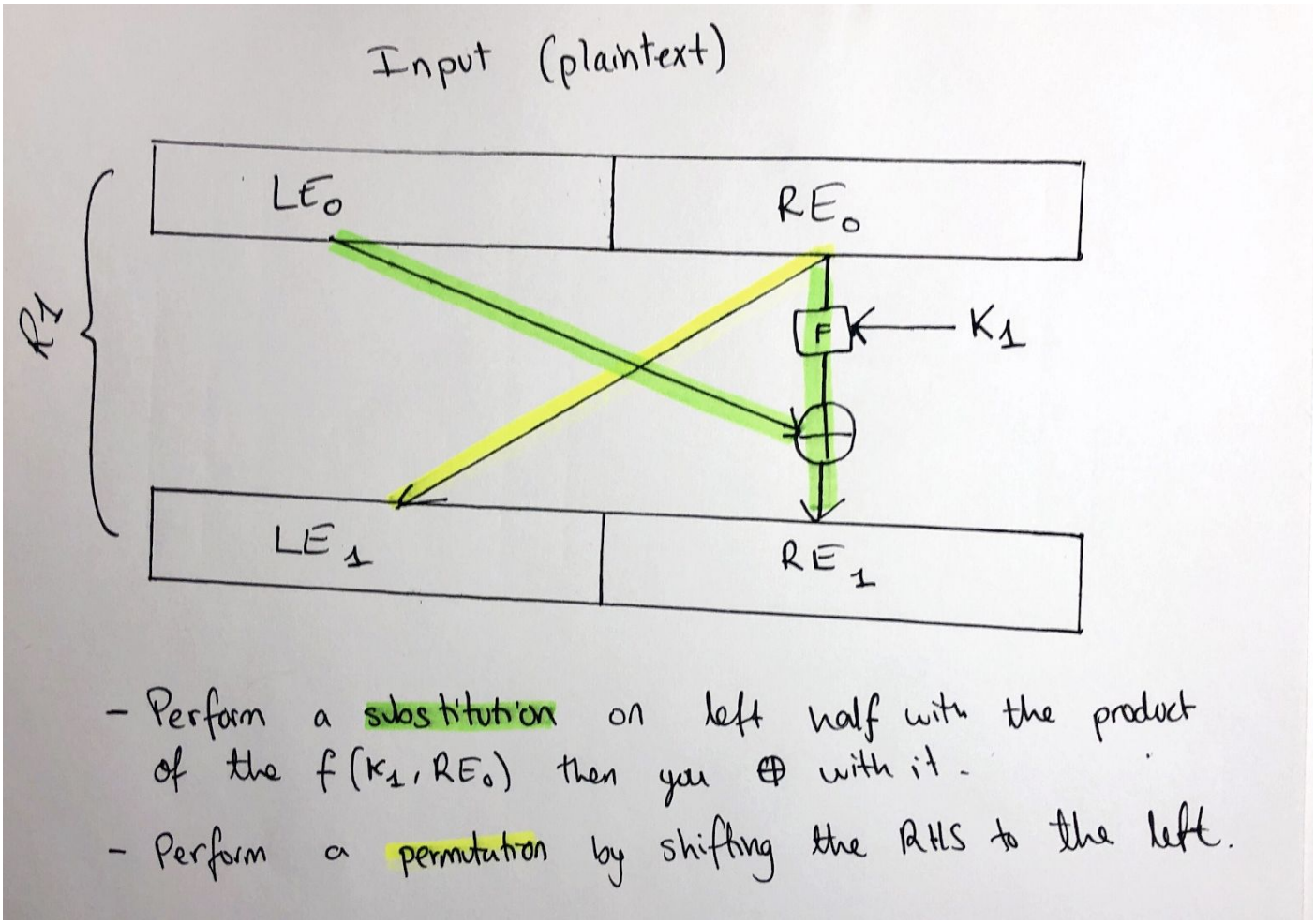
- S-Boxes "confuse" input bits.
- P-Boxes "diffuse" bits across S-box inputs

Note that permutation by itself does not diffuse.

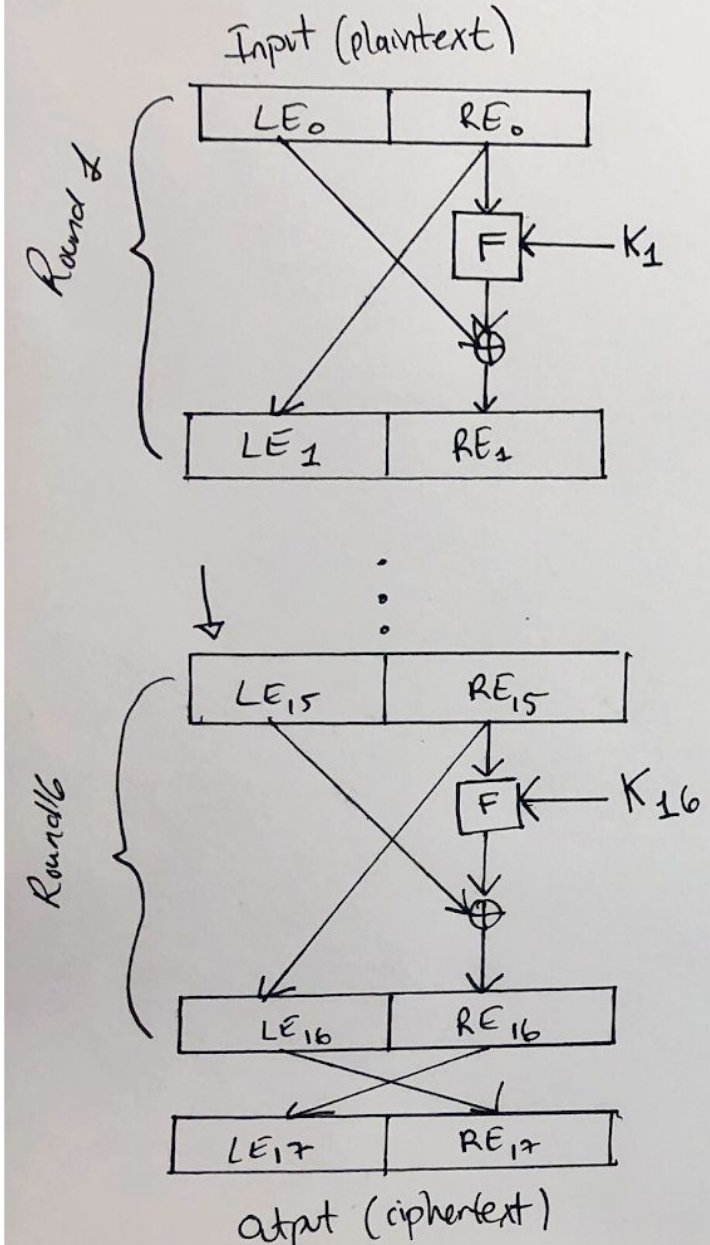
Feistel encryption / decryption (16 rounds)

1. We split our input block into two halves. The LHS and RHS.
2. We go through multiple rounds in which:
    a. We perform a substitution on the left data half.
    b. We perform a permutation by swapping halves.
    c.

F - feistel round function -> $F(K_{i+1}, RE_i)$: w bits x y bits -> w bits



FEISTEL ENCRYPTION



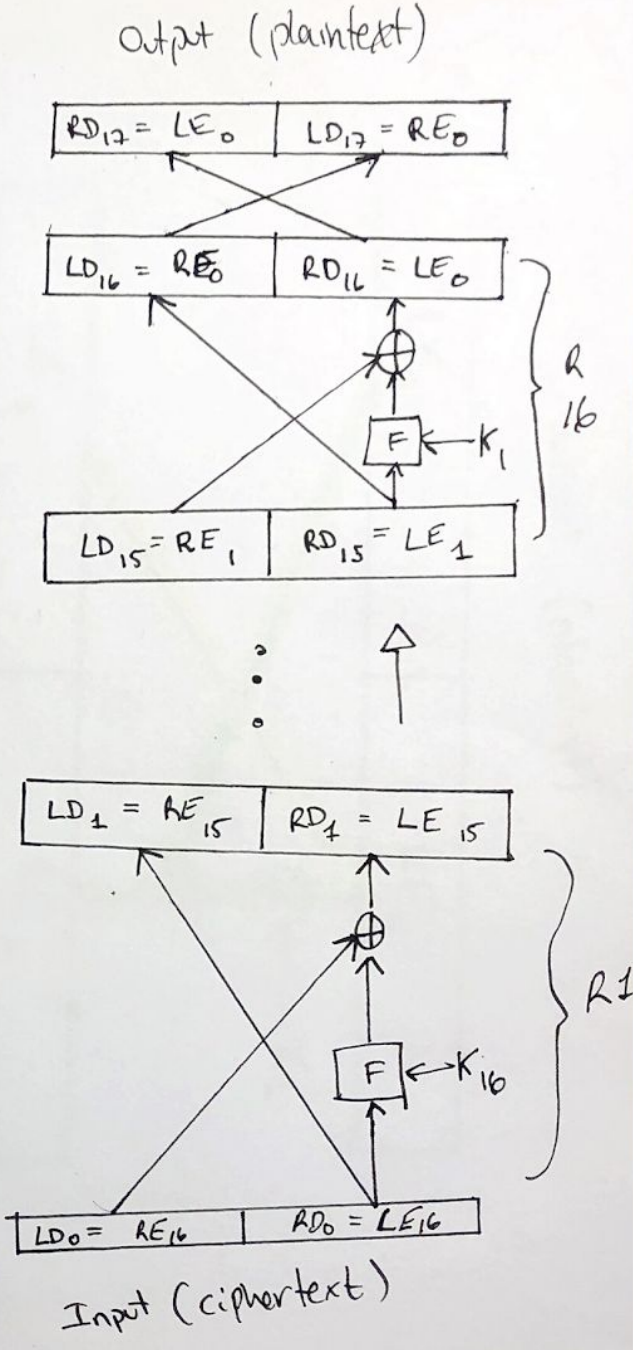
KEY:

F - Function

⊕ - XOR

RULES:

1. $17 - E_{round} = D_{round}$
2. During rounds:
   $LD_i = RE_j \mid RD_i = LE_j$
   $i \neq j = 16$
3. Dec is a horizontal flip of Enc (Enc always on RHS)
4. Enc K = Round #
   Dec K = 17 - Round #



Feistel Cipher: Parameters and design features

Block Size:

- The larger the more secure but it reduces encryption/decryption speed.
- The more the diffusion, the more secure.
- Traditionally 64 bits but now AES uses 128-bit block size

Key Size:

- The larger the more secure but it reduces encryption/decryption speed.
- Greater security achieved by greater resistance to brute force attacks and greater diffusion.
- 128 bit keys are what is used.

Number of rounds:

- Single round offers inadequate security
- Multiple rounds offer increasing security - the more you confuse things the better.

Subkey generation algorithm:

Have a very large key and you split to subkeys to generate smaller keys.

- Greater complexity means greater resistance to cryptanalysis.

Round function F :

- Greater complexity means greater resistance to cryptanalysis.

The popularity of this cipher is that given very simple parameters it produces a very strong way of hiding information, making it easy to buy hardware or a piece of software that does this.

- **Intermediate value** of decryption process

$$LD_{16-i} \parallel RD_{16-i} = RE_i \parallel LE_i$$

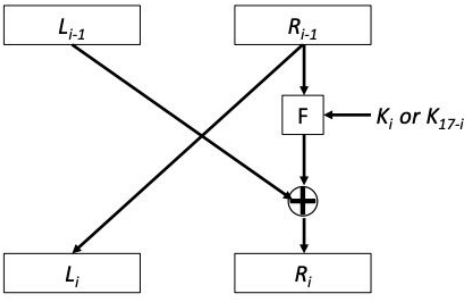
|| - means in parallel with

equal to corresponding value of encryption process with two halves of value swapped

$$LE_i \parallel RE_i$$

Encryption and decryption are the same, which is what makes it so easy to implement, what changes is the input and the key used.

One of the strengths of the Feistel cypher is that each round of encryption or decryption has the following general shape



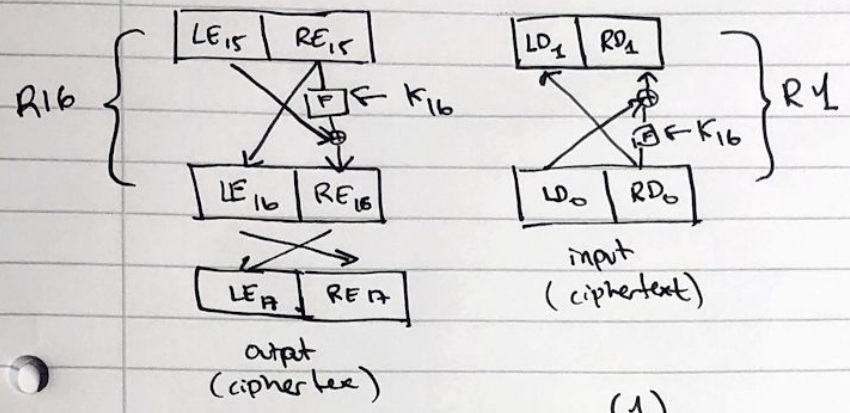
where the only difference is that

- at encryption round i, the key K_i is used,
- at decryption round i, the key K_{17-i} is used (for decryption, we take the keys in reverse order).

Showing equality between decryption output and 32-bit swap of encryption input

Show that output of decryption round 1 is equal to the 32-bit swap of encryption round 16.

Prove: $LD_1 \| RD_1 = RE_{15} \| LE_{15}$



Let's assume: (1) $LD_0 = RE_{16}$ and (2) $RD_0 = LE_{16}$

On the encryption side:

by definition:

$LE_{16} = RE_{15}$ (3)

$RE_{16} = LE_{15} \oplus F(RE_{15}, K_{16})$ (4)

On the decryption side:

$LD_1 = RD_0 = LE_{16} = RE_{15}$ (2), (3) (5)

$RD_1 = LD_0 \oplus F(RD_0, K_{16})$

$= RE_{16} \oplus F(RD_0, K_{16})$ (1)

$= LE_{15} \oplus F(RE_{15}, K_{16}) \oplus F(RD_0, K_{16})$ (4)

$= LE_{15} \oplus F(RE_{15}, K_{16}) \oplus F(RE_{15}, K_{16})$ (5)

$= LE_{15} \oplus 0$ (see below)

$= LE_{15}$ (see below)

Following XOR properties

$a \oplus a = 0$ $\therefore F(RE_{15}, K_{16}) \oplus F(RE_{15}, K_{16}) = 0$

and

$b \oplus 0 = b$ $\therefore LE_{15} \oplus 0 = LE_{15}$

This can be done for every single round, but it is crucial that we know that

Decryption Round = 17 - Encryption Round

Generally:
Thus we can always prove that for 1 <= i <= 16:

- The output of decryption round 16 - i is equal to the 32 bit-swap of the input to encryption round i+1.

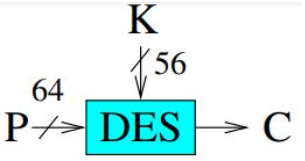
$$LD_{16-i} \parallel RD_{16-i} = RE_i \parallel LE_i$$

$$\text{for } 1 \leq i \leq 16$$

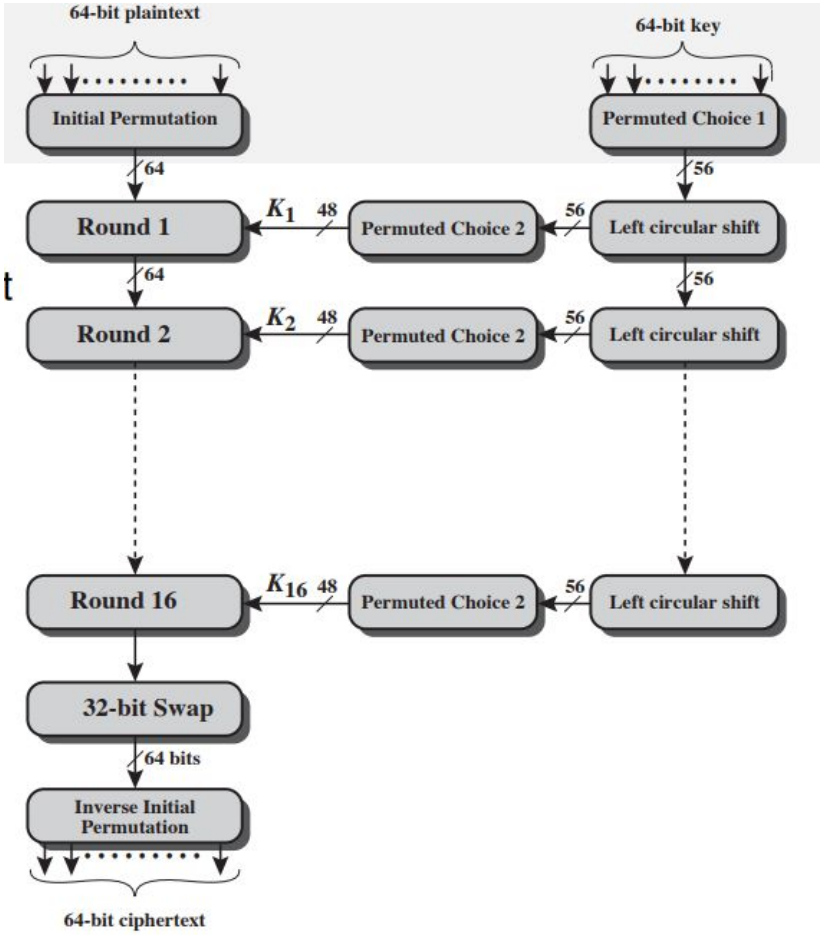
Data Encryption Standard (DES)

There was some controversy in the key length, the key is 56 bits rather than the 128 used in other ciphers at the time.

Triple DES has been used to overcome the short key length and is widely used now.



DES Overall Scheme



Block size: 64 bits
Plaintext and Ciphertext space: M = C = $\{0,1\}^{64}$

Input:

- Plaintext: 64 bits
- Key: 64 bits (but only 56 are used)

Output:

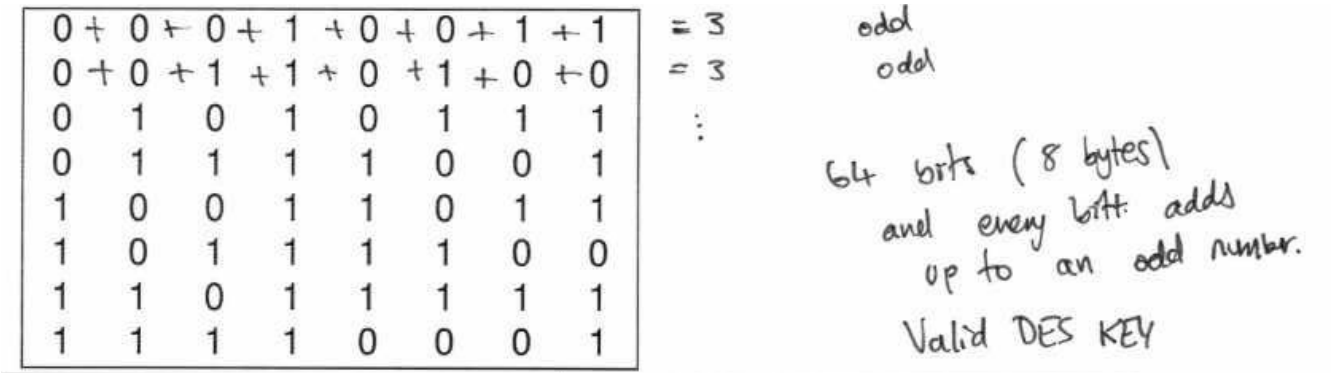
- 64 bits ciphertext

From the Master Key, you generate all the different round keys and from that you perform an extended Feistel Cipher.

Permuted Choice: operation that reduces our key (throws away 8 bits)

Valid DES Keys

If a 64-bit DES key is divided into eight bytes, then the sum of the eight bits of each byte is odd.



Therefore the key space:

$$\mathcal{K} = \{(b_1, \ldots, b_{64}) \in \{0,1\}^{64} \mid \sum_{i=1}^{8} b_{8k+i} \equiv 1 \mod 2,\ 0 \leq k \leq 7\}$$

The key space is the space of all the sequences of 64 bits where each element is a 0 or a 1 such that the sum from 1 to 8 [bit number in row] and k 0 to 7 [represents column number], the sum of b_{8k+1} (represents a row) is equal to 1 mod 2.

DES Processing Plaintext

1. Initial Permutation (**IP**) rearranges bits to produce **permuted input**.
2. 16 rounds of both permutation and substitution functions.
    a. The output of the 16th round is 32-bit swapped to produce the **pre-output**.
    b. Pre-output is passed through **IP^{-1} (inverse initial permutation)** to produce the ciphertext.

Note: IP and IP^{-1} have no real cryptographic significance, they were just included to facilitate the use of the algorithm with the hardware in the 1970s.

If the property that the bits add up to an odd number is satisfied, then we can remove the last column because we know that the first 7 bits of each byte will determine the value of the last bit.

EXAM: has sometimes asked the mathematical formula, showed you some DES keys and asked if it is a valid key or not.

Producing Subkeys

We input a 64-bit key but we use a 56-bit key. The transformation occurs via a permutation function which strips away the 8 bits of the last column of the key.

From this 56-bit key we create each subkey K_i which is produced by a combination of a **left circular shift** and a **permutation**.

- The permutation function is the same in every round but produces a different output given the shifts of bits of the key.

The Initial and Inverse Initial Permutation

To use DES, you have to use a set of fixed tables. The initial permutation and inverse initial permutation move bits around in a certain way. They contain numbers 1 to 64 which represent every bit of the plaintext.

(a) Initial Permutation (IP)

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

(b) Inverse Initial Permutation (IP^{-1})

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

After the Initial permutation, the first bit of the plaintext will actually be on the last column, 5th row down. It's just a way of saying take bit 1 and put it here, take bit 2 and put it here and so on.

Consider 64-bit input M (left) and permutation $\mathrm{IP}(M)$ (right).

M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
M_9	M_{10}	M_{11}	M_{12}	M_{13}	M_{14}	M_{15}	M_{16}
M_{17}	M_{18}	M_{19}	M_{20}	M_{21}	M_{22}	M_{23}	M_{24}
M_{25}	M_{26}	M_{27}	M_{28}	M_{29}	M_{30}	M_{31}	M_{32}
M_{33}	M_{34}	M_{35}	M_{36}	M_{37}	M_{38}	M_{39}	M_{40}
M_{41}	M_{42}	M_{43}	M_{44}	M_{45}	M_{46}	M_{47}	M_{48}
M_{49}	M_{50}	M_{51}	M_{52}	M_{53}	M_{54}	M_{55}	M_{56}
M_{57}	M_{58}	M_{59}	M_{60}	M_{61}	M_{62}	M_{63}	M_{64}

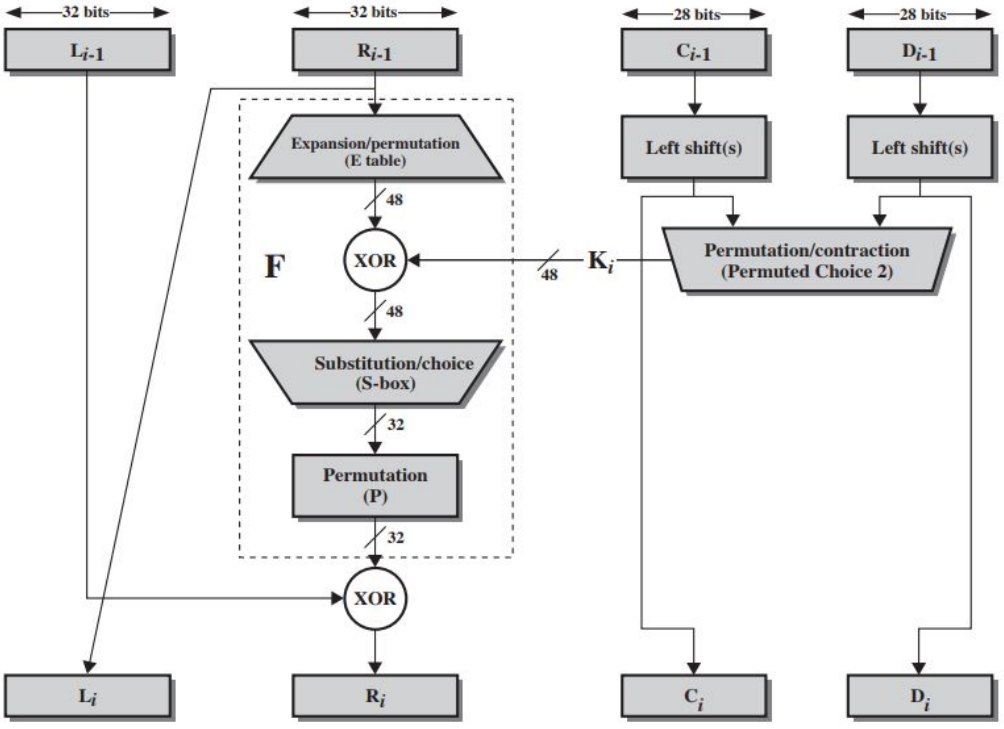
M_{58}	M_{50}	M_{42}	M_{34}	M_{26}	M_{18}	M_{10}	M_2
M_{60}	M_{52}	M_{44}	M_{36}	M_{28}	M_{20}	M_{12}	M_4
M_{62}	M_{54}	M_{46}	M_{38}	M_{30}	M_{22}	M_{14}	M_6
M_{64}	M_{56}	M_{48}	M_{40}	M_{32}	M_{24}	M_{16}	M_8
M_{57}	M_{49}	M_{41}	M_{33}	M_{25}	M_{17}	M_9	M_1
M_{59}	M_{51}	M_{43}	M_{35}	M_{27}	M_{19}	M_{11}	M_3
M_{61}	M_{53}	M_{45}	M_{37}	M_{29}	M_{21}	M_{13}	M_5
M_{63}	M_{55}	M_{47}	M_{39}	M_{31}	M_{23}	M_{15}	M_7

The Inverse initial permutation will indicate which bit is our first, which our second and so on and so forth. Example: Say our 32-bit swapped output is the following.

M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
M_9	M_{10}	M_{11}	M_{12}	M_{13}	M_{14}	M_{15}	M_{16}
M_{17}	M_{18}	M_{19}	M_{20}	M_{21}	M_{22}	M_{23}	M_{24}
M_{25}	M_{26}	M_{27}	M_{28}	M_{29}	M_{30}	M_{31}	M_{32}
M_{33}	M_{34}	M_{35}	M_{36}	M_{37}	M_{38}	M_{39}	M_{40}
M_{41}	M_{42}	M_{43}	M_{44}	M_{45}	M_{46}	M_{47}	M_{48}
M_{49}	M_{50}	M_{51}	M_{52}	M_{53}	M_{54}	M_{55}	M_{56}
M_{57}	M_{58}	M_{59}	M_{60}	M_{61}	M_{62}	M_{63}	M_{64}

Given the IP^{-1} table, our first bit of ciphertext will be M_{58} followed by M_{50}, M_{42} …

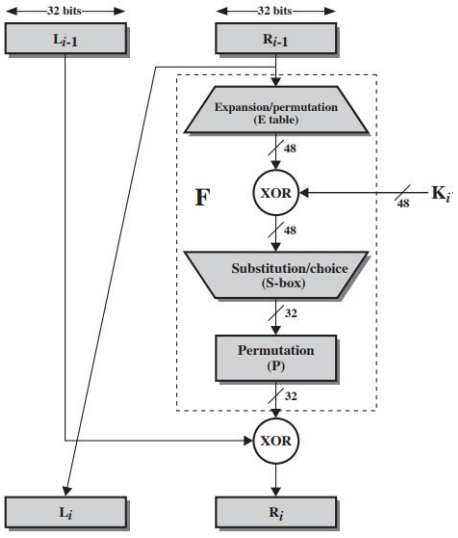
DES Encryption: details on a single round



- **Round Key K_i is 48 bits**

As we can see it is very similar to a feistel cipher. Essentially in DES we are:

- Changing the order (E table)
- Substituting (S-box)
- Changing order (P)



Feistel Box

Given our split 64 bit block, we input 32 bits into the Feistel Function but we are XORing it with a key that is 48. We use the **Expansion/Permutation Table (E Table)** to expand our input so that it can be XORd. Then after that, we have to decrease the number of bits as we want our output to be 32 bits, thus we use a **Substitution/choice Box (S-box)** which transforms the XORed key and input.

Expansion/Permutation Table (E Table)

Given that we need an input that is 48-bits like our Key, we must expand our input in the following way. We just duplicate 16 of the bits of our input. We apply this by replicating the input repeating the bits which are in the two columns on the sides. That is, we will place the first bit of our 32-bit input in the second position of the first row and in the last position of the last row and so on…

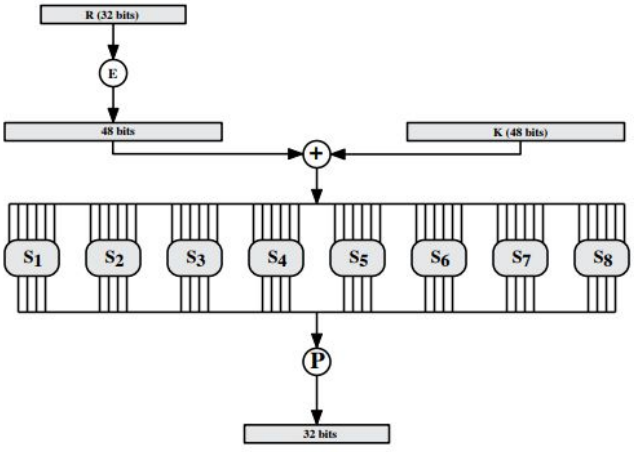
We could place the bits at the end but by putting them in front and in the back we achieve a permutation.

(c) Expansion Permutation (E)

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Substitution Box (S-box)

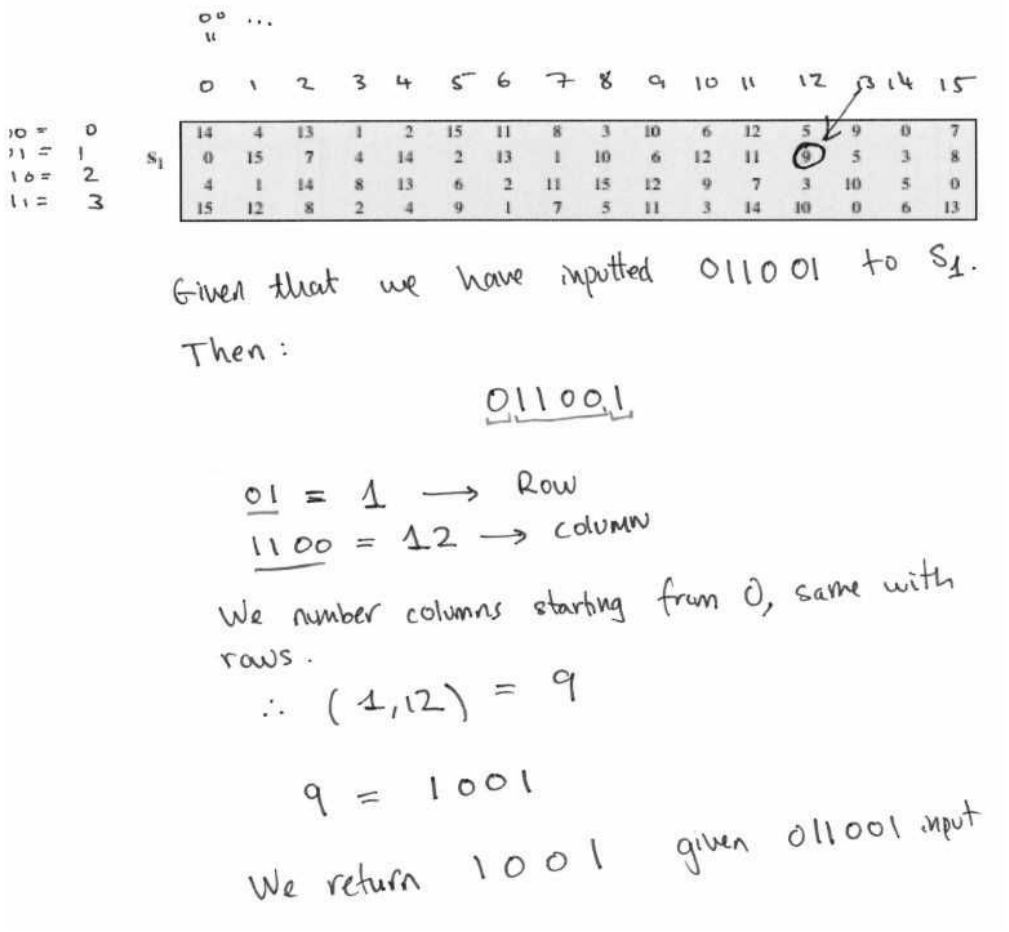
After we have produced our 48 bits and XORed them with our subkey, we then have to reduce that output to 32-bits. To do so we use an S-box, and s-box contains 8 s-boxes which take an input of **6 bits** and output **4 bits**. Which 4 bits it outputs are dependent on the box and can be looked up in tables. Therefore 4 x 8 = 32 bits which are outputted, ready to be XORed with the old left hand side.



The way in which you substitute bits is the following:

- Given our 6 bit input (*ie 011001*), we use the **first** and **last** bit to give us the numerical value of the row. (*01 = 1st row*).
- With the other 4 bits (*1100*) in between we find out the value of the column (*1100 = 12th column*).
- We then find the number in our box which represents that row and that column. Then we return the bit representation of that number in 4 bits.

Example:



EXAM: he has asked in the past and is likely to ask again how the different S-Boxes are used.

Permutation Function (P)

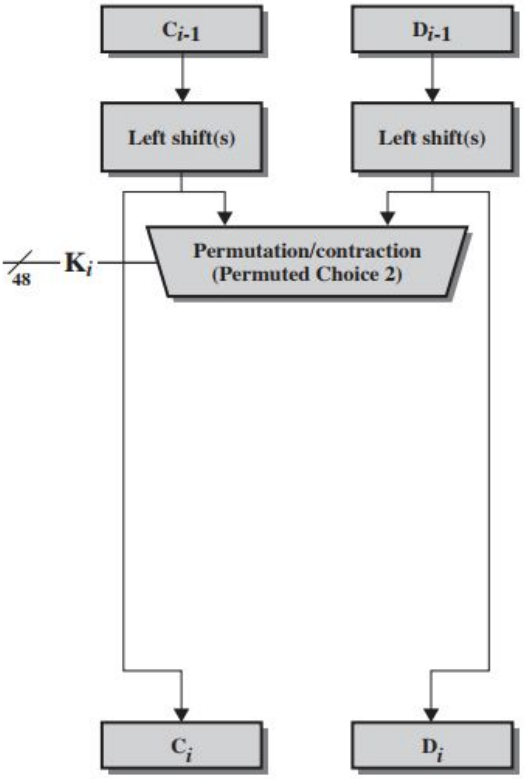
After passing through a substitution function (s-box) which produces a 32-bit output, it is then permuted using a **Permutation Function P.**

(d) Permutation Function (P)

16	7	20	21	29	12	28	17
1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9
19	13	30	6	22	11	4	25

Key Generation

To calculate the round key we split the key, we then perform a **left shift** on one part, a **left shift** on the other, we then propagate this into a **Permutation/contraction (Permuted Choice 2)** to generate the key.



Note that in principle that the sum of the bits in a byte of a DES key is odd is just a choice, it could've been that it was even. The important thing is that given the other 7, we are able to infer the last one.

Ignore the 8th bit of every row

From a 64-bit key inputted, we need 56 bits, therefore we ignore the 8th bit of every row (or every 8th bit).

We keep the shaded parts.

(a) Input Key

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Permuted Choice One

Then we take the remaining bits and change their order according to the function **Permuted Choice One.**
Note that bit 8 is not in the list (it is one of the ones we dropped when we dropped bits to trip the size of our key). 16, 24, 32 … are not there either.

Once we have rearranged the bits, we then split the key in two. Resulting in C_{i-1} and D_{i-1}

(b) Permuted Choice One (PC-1)

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Circular Shifts

We then perform circular **left** shifts, we move bits to the left that fall out and come up on the other side. We will shift them 1 or 2 bits depending on the round number as shown in the table.

Round Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Bits Rotated	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

These shifted values serve as input to the key **next round** and to the **Permuted Choice Two** which produces a 48-bit output.

Permuted Choice Two

We again reorder bits according to the table. This function produces a 48-bit output that is then fed into the Feistel Function. Note that it omits some bits to decrease the size of the key. The output of the table will be the key.

(c) Permuted Choice Two (PC-2)

14	17	11	24	1	5	3	28
15	6	21	10	23	19	12	4
26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40
51	45	33	48	44	49	39	56
34	53	46	42	50	36	29	32

DES Decryption

It uses the same algorithm as encryption, except the application of the subkeys is reversed. We apply subkey 16 first.

Note, if the plaintext that we input to DES is less than 64 bits, then we pad, if it is more, we split it into blocks.

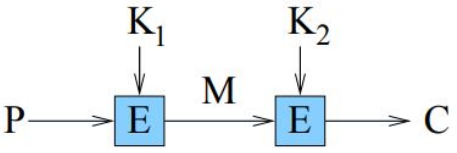
Security of DES

DES has been broken by exhaustively searching the key space (brute-force attack). And it was proven to be broken in 1993 in 7h.

To use it, we must increase its security.

Increasing DES Security: Double Des (2DES)

Idea: Perform two DES encryptions



What if we perform DES twice with two keys? Does my security increase? Is it equivalent to performing DES with a 112 bit key?

The answer is no since double DES is subject to an attack called **Meet-in-the-Middle.** We are actually searching a space of 56 bits twice, rather than one that is 112.

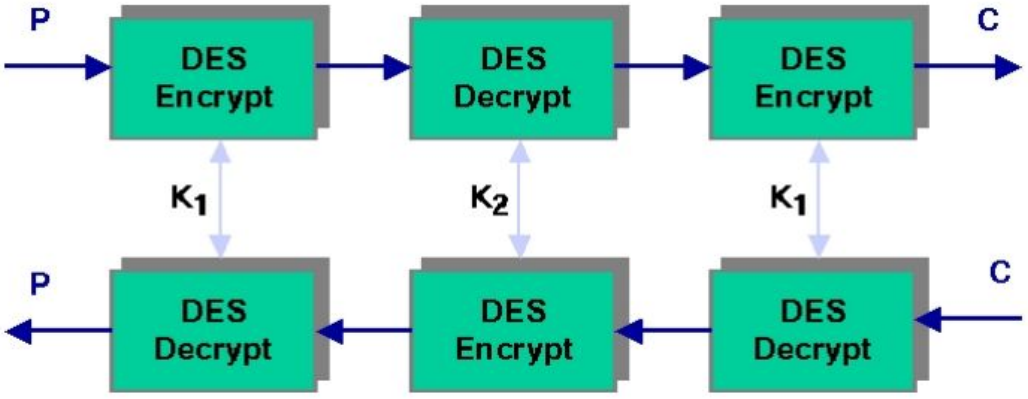
Meet-in-the-Middle:

- For $C = E(K_2, E(K_1, P))$
- Let $X = E(K_1, P) = D(K_2, C)$
- Given that I know C and P. (known plaintext and ciphertext attack)
- We can then generate all the 2^{56} keys and encrypt P to find X
- Store X in a table and sort it.
- Then I decrypt C with all the keys and find where X matches.
- Each coincidence is a candidate solution which we can then validate with an additional plain/cipher-text pair.

We've generated 2^{56} encryptions and 2^{56} decryptions but these are of the order 56 which means that we have a $2 * 2^{56}$ number of operations which is of the order of 57. Not in the order of 128.

Increasing DES Security: Triple DES (a.k.a. 3DES)

Standard 3DES is done with 3 different keys K_1, K_2



Remember that decryption is just a function, we can decrypt whatever it does not have to produce and intelligible solution.

We do not know if it can be broken, there is no practical attack and to brute-force search with 2^{112} operations.

This is the standard triple DES because it is compatible with DES. To make it compatible we set $\mathbf{K_2 = K_1}$ then the first two Encryption and Decryptions cancel themselves out and we encrypt under DES, the converse for decryption.

We can also do DES with 3 different keys, K1, K2, K3.

- Effective key length is 168 bits.

Possible Q: How does 3DES or more DES is more secure?

AES

AES is a **symmetric block cipher** intended to replace DES for commercial operations. It uses a 128-bit block size and a key size of 128, 192 or 256 bits.

· AES does not use a **Feistel structure**, rather each round consists of 4 separate functions: *byte substitution, permutation, arithmetic operations over a finite field and XOR with a key.*

AES Encryption

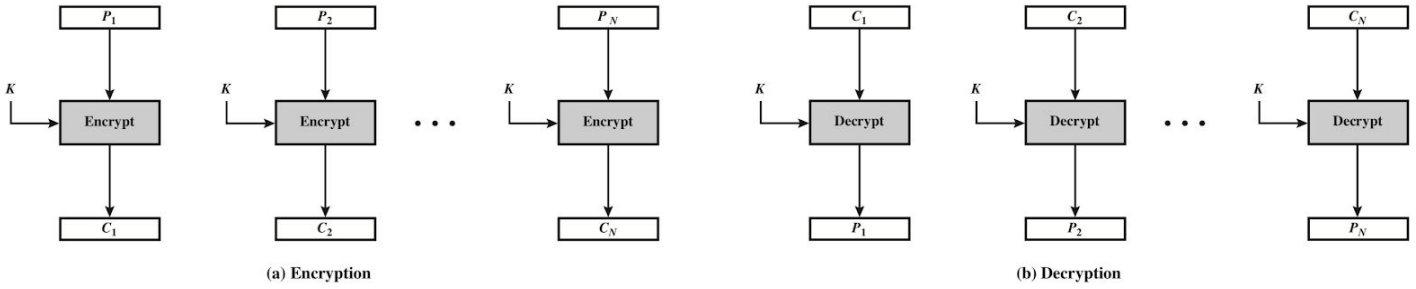
· Plaintext block size of 128 bits (16 bytes)
· Key length can be 128, 192 or 256 bits
· Cipher consists of N rounds, depending on key length

Block cipher modes of operation: ECB, CBC, CFB, OFB ,CTR

When we don't have a 64-bit input but we have more than that we split the blocks. The way in which we split these blocks and how we handle them are called modes of operation.

Mode of operation: a technique for enhancing the effect of a cryptographic algorithm or adapting the algorithm for an application, such as applying a block cipher to a sequence of data blocks or data stream.
We will only look at the first 4.

Electronic Codebook (ECB)



Is the simplest, we encrypt every single block separately and we decrypt it separately.

- It requires the last block to be padded if it's a partial block.
- Each Pi has a value that is substituted, like a codebook, hence the name.

- You should not use it for messages that are longer than 64 bits. Because if the same block appears twice it will create the same ciphertext, it does not give us additional security.
    - If a message is highly structured, a cryptanalyst would be able to exploit regularities.

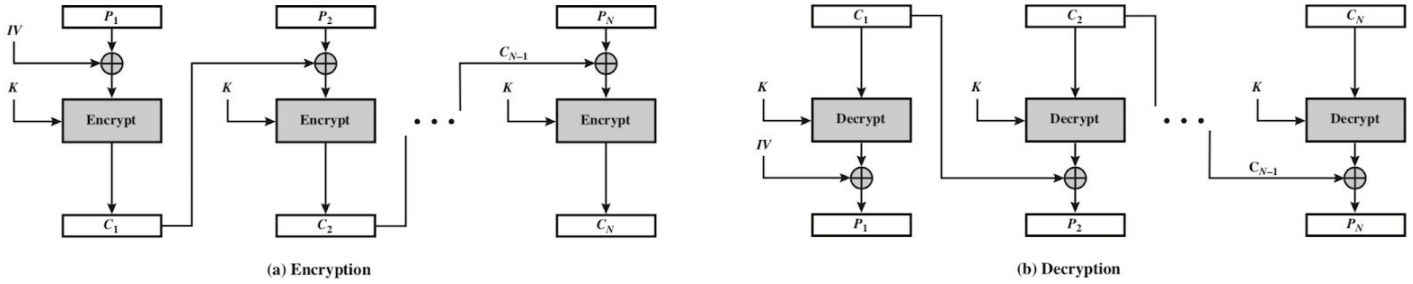
+ Ideal for short amount of data, like an **encryption key** (ie to transmit DES and AES keys securely). For example we can send the next DES key encrypting it with the current key to be able to encrypt/decrypt the next blocks. This is done often since breaking a 64-bit key is easy therefore you must refresh the key often.

Cipher-block Chaining (CBC)

To overcome issues with ECB, we must make sure that if we are encrypting the same block, it produces a different ciphertext. **Cipher input is XOR of current plaintext block with previous ciphertext block.**

For the first encryption, we do not have a C_0 so we need to create it. To do so we use an Initialisation Vector (IV) which is of the length of our block size and XOR is with the first plaintext.

Encryption:	Decryption:
$C_1 = E(K, IV \oplus P_1)$	$P_1 = D(K, C_1) \oplus IV$
$C_i = E(K, C_{i-1} \oplus P_i)$	$P_i = D(K, C_i) \oplus C_{i-1}$



- Requires the last block to be passed to the full number of bits in a block.

The reason why it's called cipher-block chaining is that you create a chain of block when you use the new ciphertext.

Proving for **correctness**, checking that the encryption is really the inverse of the decryption.

$$\begin{aligned} P_i &= D(K, C_i) \oplus C_{i-1} \\ &= D(K, E(K, C_{i-1} \oplus P_i)) \oplus C_{i-1} \\ &= (C_{i-1} \oplus P_i) \oplus C_{i-1} \\ &= P_i \end{aligned}$$

Given that it's symmetric encryption in STEP 2, E and D cancel themselves out.
EXAM: This makes a very good question.

Initialisation Vector IV must:

- Be the same size as the data block.
- Known to both sender and receiver but be unpredictable by a third party. (we can use ECB for this)
- Should be protected against unauthorized changes. If it had a very defined structure and was easy to know, an attacker could generate the input to the encryption which defeats the purpose, equivalent to ECB.
- In this case we see that we might want to use ECB to send the key and the IV.

Properties

Repeating patterns of b bits are not exposed because each plaintext block has no fixed relationship to the plaintext block.

- Identical plaintext blocks mapped to different ciphertext.
- Chaining dependencies: C_i depends on all preceding plaintext.

- Self-synchronizing: if an error occurs (changed bits, dropped blocks) in C_i but not C_{i+1} then C_{i+2} is correctly encrypted. *For example when you are transmitting a signal, there can be noise and some bits may be flipped.*

*CBC is good to encrypt messages longer than our block size **b**.*

Converting from block cipher to stream cipher

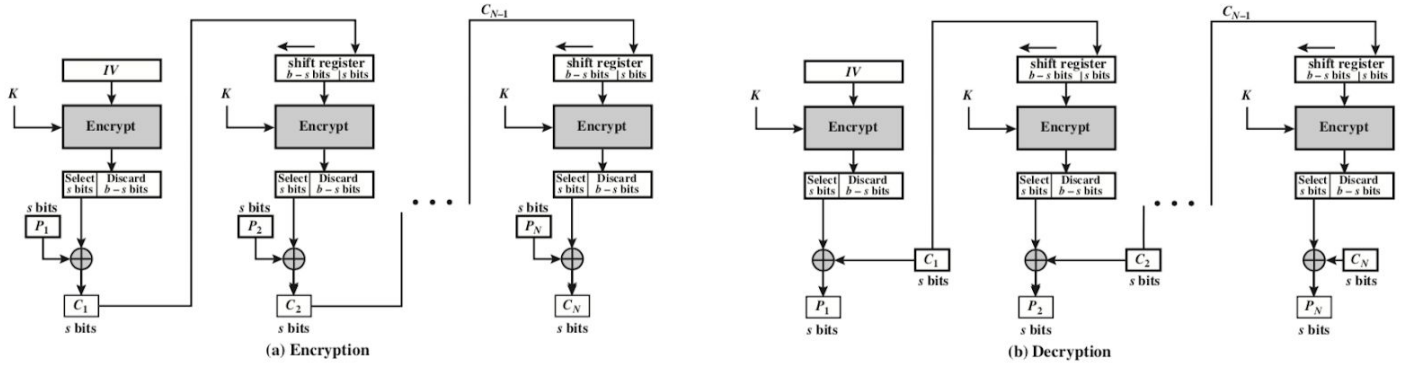
We can convert block ciphers to stream ciphers using modes CFB and OFB. This may be good because:

+ Stream ciphers don't need to pad messages.
+ Can operate in real time, each character being transmitted is encrypted and transmitted immediately.
+ We do not need that the plaintext and the ciphertext are the same length with stream ciphers as we do with block ciphers.

CFB

EXAM: in the past he has asked to draw the pictures

We divide our plaintext into **segments** rather than blocks (segments are shorter than blocks). Common value is 8.



CFB Encryption:

First round:

1. Start with the initialisation vector and encrypt it.
2. Select the most significant s bits (that is the ones that represent the higher numbers, the first s bits from the left)
3. XOR it with our plaintext

So:

$$C_1 = P_1 \oplus MSB_s(E(K, IV))$$

Other rounds:

1. Shift the previous input s bits and append s bits from the plaintext, and encrypt it.
2. Select the most significant s bits (that is the ones that represent the higher numbers, the first s bits from the left)
3. XOR it with our plaintext

CDB Decryption:

We use the same process as encryption but here we produce the Plaintext because we XOR the output of the encryption with the ciphertext. We also feed that into the next round.

Encryption:	Decryption:	
$I_1 = IV$	$I_1 = IV$	
$I_i = LSB_{b-s}(I_{i-1}) \parallel C_{i-1}$	$I_i = LSB_{b-s}(I_{i-1}) \parallel C_{i-1}$	for $i = 2, \ldots, N$
$O_i = E(K, I_i)$	$O_i = E(K, I_i)$	for $i = 1, \ldots, N$
$C_i = P_i \oplus MSB_s(O_i)$	$P_i = C_i \oplus MSB_s(O_i)$	for $i = 1, \ldots, N$

Although CFB can be viewed as a stream cipher, it does not conform to the typical construction of a stream cipher:

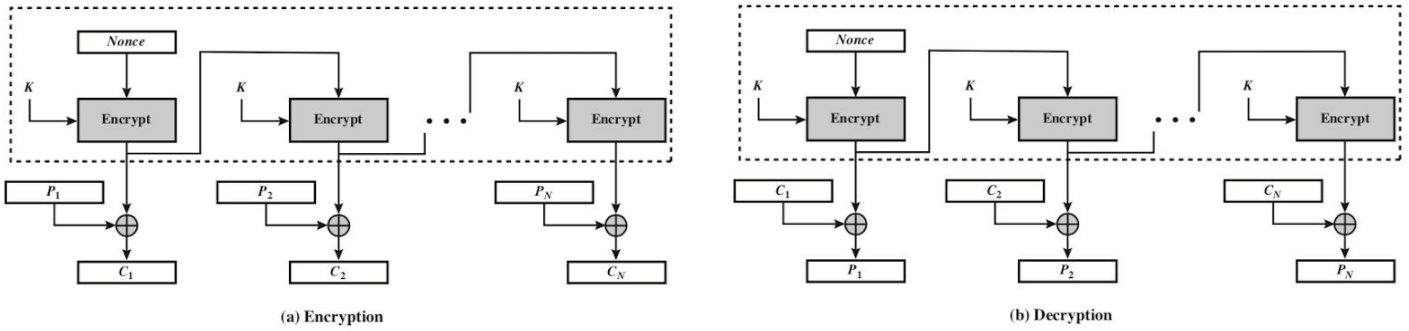
- Typical stream cipher takes as input some initial value and a key and generates a stream of bits, which is then XORed with plaintext bits.

CFB: stream of bits that is XORed with plaintext also depends on plaintext because we are reusing s bits from ciphertext.

Output Feedback (OFB)

Similar to CFB but **output of encryption function is fed back to shift register** in OFB (in CFB, ciphertext unit is fed back to the shift register) and **OFB operates on full blocks, not on an s-bit subset.**

Nonce: time-varying value that has a negligible chance of repeating (random variable like timestamp, sequence number or combination of these)



Encryption:	Decryption:	
$I_1 = IV$	$I_1 = IV$	a nonce
$I_i = O_{i-1}$	$I_i = O_{i-1}$	for $i = 2, \ldots, N$
$O_i = E(K, I_i)$	$O_i = E(K, I_i)$	for $i = 1, \ldots, N$
$C_i = P_i \oplus O_i$	$P_i = C_i \oplus O_i$	for $i = 1, \ldots, N-1$
$C_N^* = P_N^* \oplus MSB_u(O_N)$	$P_N^* = C_N^* \oplus MSB_u(O_N)$	

The last equation is just in case there are u bits instead of b bits. Then we take the MSB_u of the output and xor it with our plaintext or ciphertext.

- For a given Key and IV, the output used to XOR our stream P*i* is fixed.
- If 2 different messages had an identical block of plaintext in identical position, an attacker could determine that proportion of O_i stream

- OFB has the structure of a typical stream cipher (but one block at a time). It generates a stream of bits as a function of an initial value and a key, then that stream is XORed with plaintext bits.

- The generated stream that is XORed with the plaintext is independent (could be precomputer)
- We can use OFB with a shift register as well as with CFB.

Feedback based modes of operation

All the NIST-approved block cipher modes (except ECB) involve feedback.

Alice and Bob, before they can communicate must agree on three things: **the algorithm used, key and IV.**

RULE, for every mode of operation that feedback something between rounds, we ALWAYS feed back the ciphertext in both encryption and decryption. Except for Output Feedback.

RULE2, the two that have feedback in their word, cipher feedback and output feedback only use encryption functions, the rest use both encryption and decryption.

Key Distribution

Key distribution problem: we have issues to distribute keys securely (given that the security of our system depends on this). Often secure system failures occur by breaking the key distribution scheme.
We have a couple alternatives:

- A can send the key physically to B
- If A and B have communicated previously, they can use previous key to encrypt a new key
- Third party can select and deliver key A to B
- If A and B have secured communications with party C, C can relay key between A and B.

Typically we will use a hierarchy of keys

- We will have a **session key**, used for a short lifetime
- **Master key** used to encrypt session keys shared by users and key distribution centers

This comes down to a mutual trust, does alice trust bob, do they trust the first party? Public key cryptography helps overcome this need for trust.

Public Key Cryptography

Public key cryptography was born to solve two problems, key distribution and how to agree on which key to use. In public key cryptography, we have two keys a private and a public. Given that we encrypt something with our public key, it should then be decrypted with our private key.

> Let $\{E_e \mid e \in \mathcal{K}\}$ and $\{D_d \mid d \in \mathcal{K}\}$ form an encryption scheme. Consider transformation pairs (E_e, D_d) where knowing E_e it is infeasible, given $c \in \mathcal{C}$, to find an $m \in \mathcal{M}$ such that $E_e(m) = c$.

Meaning even if we know the public key (encryption key) it should be **infeasible** given the ciphertext to recover the private key (d).

Therefore, E_e is a trapdoor one-way function meaning everybody can encrypt with trapdoor d. Which means that to reverse it you need a trapdoor, a secret information a private key (d).

Syntax for PKE

Alice's public key written equivalently as K_A, K_a, KA, Ka, PU_A, PU_a, $PU(A)$, $PU(a)$,...

Alice's private key written equivalently as K_A^{-1}, K_a^{-1}, KA^{-1}, Ka^{-1}, PR_A, PR_a, $PR(A)$, $PR(a)$, ...

Symmetric vs asymmetric encryption

Conventional Encryption	Public-Key Encryption
Needed to Work: 1. The same algorithm with the same key is used for encryption and decryption. 2. The sender and receiver must share the algorithm and the key. *Needed for Security:* 1. The key must be kept secret. 2. It must be impossible or at least impractical to decipher a message if no other information is available. 3. Knowledge of the algorithm plus samples of ciphertext must be insufficient to determine the key.	*Needed to Work:* 1. One algorithm is used for encryption and decryption with a pair of keys, one for encryption and one for decryption. 2. The sender and receiver must each have one of the matched pair of keys (not the same one). *Needed for Security:* 1. One of the two keys must be kept secret. 2. It must be impossible or at least impractical to decipher a message if no other information is available. 3. Knowledge of the algorithm plus one of the keys plus samples of ciphertext must be insufficient to determine the other key.

Public Key cryptosystem

To achieve:

- Confidentiality: we can encrypt with the recipients public key, so that only he can decrypt it and read it.
- Authentication: we encrypt with our private key so it can only be decrypted with our public key, note that everyone can decrypt it at this point.
- Secrecy and authentication: we can encrypt the plaintext with our private key and encrypt it again with the recipient's public key, therefore the recipient will decrypt it with his private key (secrecy) and then decrypt it with the sender's public key (authentication).

Applications of PKE

- Encryption/Decryption
- Digital signatures: sender "signs" a message with its private key
- Key exchange: we can use PKE to send symmetric keys or to agree on one.

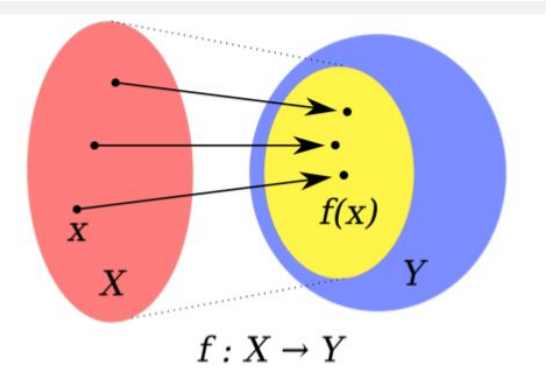
Requirements for Public-key cryptography

1. It should be easy for an agent to generate a pair of keys
2. It should be computationally easy to take the public key of someone and encrypt a message
3. It should be computationally easy for a receiver to decrypt C using the correct private key
4. It must be computationally infeasible to determine the private key if you know the public key
5. (Not always true) the two keys can be applied in each other [that is that the two keys are a perfect inverse of each other]

Algorithm	Encryption/Decryption	Digital signature	Key exchange
RSA	Yes	Yes	Yes
Elliptic Curve	Yes	Yes	Yes
Diffie-Hellman	No	No	Yes
DSS	No	Yes	No

Function Terminology

f : X -> Y is a function from the set X (f's domain) to set Y (co-domain [the possible values that come out])



The **image** [actual values that come out] of f, is the subset of f's codomain that is the output of f in a subset of its domain. In other words, is the element that x is mapped to.
Formally:

- Image of $x \in X$ under f is $f(x) = y$ (value of f when applied to x).
- Image of a subset $A \subseteq X$ under f is subset $f[A] \subseteq Y$ defined by

$$f[A] = \{y \in Y \mid y = f(x) \text{ for some } x \in A\}$$

The **inverse image (preimage)** of a subset (B) of the codomain of a function f, is the set of all elements of the domain that map to the members of the subset (B).

$$f^{-1}[B] = \{x \in X \mid f(x) \in B\}$$

One-way function

A function which is easy to compute for all x, but infeasible or hard to compute the inverse of it f^{-1}
Just like dropping a mug, gravity is a function that breaks the mug, putting it back together is very difficult.

- Easy: defined to mean that can be solved in polynomial time (if you input n bits, the time to compute the is proportional to n^a, where a is a fixed constant).
- Infeasible: the effort necessary to solve the problem grows quicker than polynomial time as function input size increases. Ie. time to compute function is proportional to 2^n where n is the input length

Examples for humans:

- Square root
- Modular cube roots

Trapdoor one-way function

A trapdoor one-way function is easy to calculate in one direction and infeasible to calculate in the other direction unless certain additional information is known. With additional info, inverse can be calculated in polynomial time.

Hence a trapdoor one-way function is a family of invertible functions f_k such that computing:

$Y = f_k(X)$ is easy if k and X are known
$X = f_k^{-1}(Y)$ is easy if k and Y are known
$X = f_k^{-1}(Y)$ is infeasible if Y is known but k is not known

Example: computing modular cube roots is easy when p and q are known.

Public-key cryptanalysis

Brute-force attacks are possible

- Using large keys counter this. We must find the correct trade-off between the complexity of encryption and decryption. Normally 1024 or 4096 bits long.

Computing private key from public key
We have no proof that this attack is infeasible mathematically
Probably-message attack
If the message is short (like a 56-bit DES key) and with someone's public key. I can then create all the possible messages of a space of 56-bits, apply the public key to it and check if they correspond to the ciphertext.
Solution: append random bits to M before encryption to make it longer.

RSA

Number Theory

- Natural Numbers: N = {0, 1, 2, ...}
- Integers: Z = {0, 1, -1, ...}
- Primes: P = {2, 3, 5, 7, ...}

A **factor** is a number with which we can write a product of another number. Ie. a x b = c, a and b are factors.
Multiplying numbers is easy but factoring them is complicated.

The **Prime factorization** of a, is writing it as a product of powers of primes:

$$a = \prod_{p \in \mathcal{P}} p^{a_p} = 2^{a_2} \times 3^{a_3} \times 5^{a_5} \times 7^{a_7} \times 11^{a_{11}} \times \ldots \quad \text{where } a_p \in \mathbb{N}$$

For any particular value of a, most of the exponents a_p will be 0, e.g.,

$$\begin{aligned} 91 &= 7 \times 13 \\ 3600 &= 2^4 \times 3^2 \times 5^2 \end{aligned}$$

Divisors

$a \neq 0$ **divides** b (written $a \mid b$) if there is an m such that $m \times a = b$.

- Examples: $3 \mid 6$ and $7 \mid 21$.

a **does not divide** b (written $a \nmid b$) if there is no m such that $m \times a = b$.

- Examples: $3 \nmid 7$, $3 \nmid 10$ and $7 \nmid 22$.

Relatively Prime Numbers & Greatest Common Divisor

Greater common divisor gcd(x,y): is the greatest number that divides both numbers.
Relatively prime: two numbers are relatively prime if they have no common divisors/factors apart from 1 (gcd(a,b) = 1)

8 and 15 are relatively prime given that:

- Factors of 8 are: 1,2,4,8
- Factors of 15 are: 1,3,5,15
- 1 is the only common juan

Euclid's Algorithm and Extended Euclid's Algorithm

We can use Euclid's Algorithm to compute gcd.

$$\begin{array}{lcll} \gcd(60,14) & : & 60 & = & (4 \times 14) + 4 \\ \gcd(14,4) & : & 14 & = & (3 \times 4) + 2 \\ \gcd(4,2) & : & 4 & = & 2 \times 2 \end{array}$$

The **Extended Euclid's Algorithm** computes x,y integers (can be negative) such that:

$$\gcd(a,b) = (x \times a) + (y \times b)$$

$$\text{Here } 2 = 14 - 3 \times (60 - (4 \times 14)) = (-3 \times 60) + (13 \times 14)$$

Euclid's Algorithm

gcd(a,b) = gcd(b, a mod b) for any nonnegative integer a and any positive integer b.

E.g: gcd(55, 22) = gcd(22, 55 mod 22) = gcd(22, 11) = 11

Euclid's algorithm

```
Euclid(a, b)
1 if b = 0
2 then return a
3 else return Euclid(b, a mod b)
```

Extended Euclid's Algorithm

Returns (greatest common divisor (d), x, y)

Extend Euclid's algorithm to compute integer coefficients x, y such that

$$d = \gcd(a, b) = (a \times x) + (b \times y)$$

Extended Euclid's algorithm

```
Extended-Euclid(a, b)
1 if b = 0
2 then
3    return (a, 1, 0)
4 else
5    (d', x', y') ← Extended-Euclid(b, a mod b)
6    (d, x, y) ← (d', y', x' − (⌊a/b⌋ × y'))
7    return (d, x, y)
```

where $q = \lfloor a/b \rfloor$ is the **quotient of the division** (for $a = (q \times b) + r$).

***d** is the greatest common divisor.*

Modular Arithmetics

Remainder

- $\forall a\, n.\ \exists q\, r.\ (a = (q \times n) + r)$ where $0 \leq r < n$.
  Here r is the **remainder**, which we write as

$$r = a \bmod n\,.$$

Congruent modulo

- $a, b \in \mathbb{Z}$ are **congruent modulo** n, if $a \bmod n = b \bmod n$.
  We write this as

$$a =_n b\,.$$

Modulo operator has following properties (of congruences)

- Reflexivity: $a =_n a$.
- Symmetry: If $a =_n b$ then $b =_n a$.
- Transitivity: If ($a =_n b$ and $b =_n c$) then $a =_n c$.

To understand the concept we can think of a watch. It's 6am or 6pm but both numbers (6 and 18) are congruent modulo 12. Therefore, $6 =_{12} 18$

Properties of Modulo Operator

Essentially, we can operate numbers as we do normally, given that we include mod n inside and outside every item of our equation.

$$(a \bullet b) =_n (a \bmod n) \bullet (b \bmod n) \qquad \text{for } \bullet \in \{+, -, \times\}$$

$$\text{i.e., } (a \bullet b) \bmod n = [(a \bmod n) \bullet (b \bmod n)] \bmod n$$

Example:

$$\begin{aligned} 2 &= (5 \times 6) \bmod 4 \\ &= [(5 \bmod 4) \times (6 \bmod 4)] \bmod 4 \\ &= (1 \times 2) \bmod 4 = 2 \bmod 4 = 2 \end{aligned}$$

We can use this to simplify operations when we have very large numbers.

If $a \times b =_n a \times c$ and a relatively prime to n, then $b =_n c$.

Example:

- $8 \times 4 =_3 8 \times 1$.
- 8 is relatively prime to 3.
- So: $4 =_3 1$.

Also used to simplify equations

If $a_1 =_n b_1$ and $a_2 =_n b_2$, then

$$(a_1 + a_2) =_n (b_1 + b_2) \quad \text{and} \quad (a_1 \times a_2) =_n (b_1 \times b_2)$$

- This can also be expressed as

$$[(a_1 \bmod n) + (a_2 \bmod n)] \bmod n = (a_1 + a_2) \bmod n$$

and

$$[(a_1 \bmod n) \times (a_2 \bmod n)] \bmod n = (a_1 \times a_2) \bmod n$$

- $a = q \times n + r \quad$ with $q = \lfloor a/n \rfloor$ and $0 \leq r < n$ and $r = a \bmod b$
- For any integer a, we can rewrite this as follows:
  $a = \lfloor a/n \rfloor \times n + (a \bmod n)$

This is the same method we would use to find the remainder on a calculator.

Then, for example:

- $11 \bmod 7 = 4$
- $-11 \bmod 7 = -4$ ($= 3$ when reasoning modulo 7)
- $73 =_{23} 4$
- $21 =_{10} -9$
- $147 =_{220} -73$

Q = quotient, r = remainder, n = multiple

- If $a =_n 0$ then $n \mid a$
- $a =_n b$ if $n \mid (a - b)$

- To demonstrate the last point, if $n \mid (a - b)$, then $(a - b) = k \times n$ for some k.

  So we can write $a = b + (k \times n)$.

  Therefore, ($a \bmod n$) = (remainder when $b + (k \times n)$ is divided by n) = (remainder when b is divided by n) = ($b \bmod n$).
- Then, for example:
  - $23 =_5 8$ because $23 - 8 = 15 = 5 \times 3$
  - $-11 =_8 5$ because $-11 - 5 = -16 = 8 \times (-2)$
  - $81 =_{27} 0$ because $81 - 0 = 81 = 27 \times 3$

Theorem 1:

If $a, b \in \mathbb{Z}$ are relatively prime
We claim:

There is a $c \in \mathbb{Z}$ satisfying $(b \times c) \bmod a = 1$

If that is the case, we can compute b^{-1} because $b \times b^{-1} = 1$, therefore we can find the inverse of b when working modulo a.

Proof: Extended Euclid Algorithm, there exist $x, y \in \mathbb{Z}$ such that:

$$1 = ax + by$$

1 because a and b are relatively prime
$\gcd(a,b) = 1$

Since $a \mid (ax)$, if we take the equation at mod a, then by mod $a = 1$.

$$1 = ax + by$$
$$1 = [(ax \bmod a) + (by \bmod a)] \bmod a$$
$$1 = [\ 0 \ + (by \bmod a)]$$
$$1 = by \bmod a$$

Therefore: $c = y$
Once we find the Euclidean Extended Algorithm of two relatively prime numbers, we can find out the inverse.

Fermat's Little Theorem

For a and n relatively prime and n is prime.

$$a^{n-1} \equiv_n 1$$

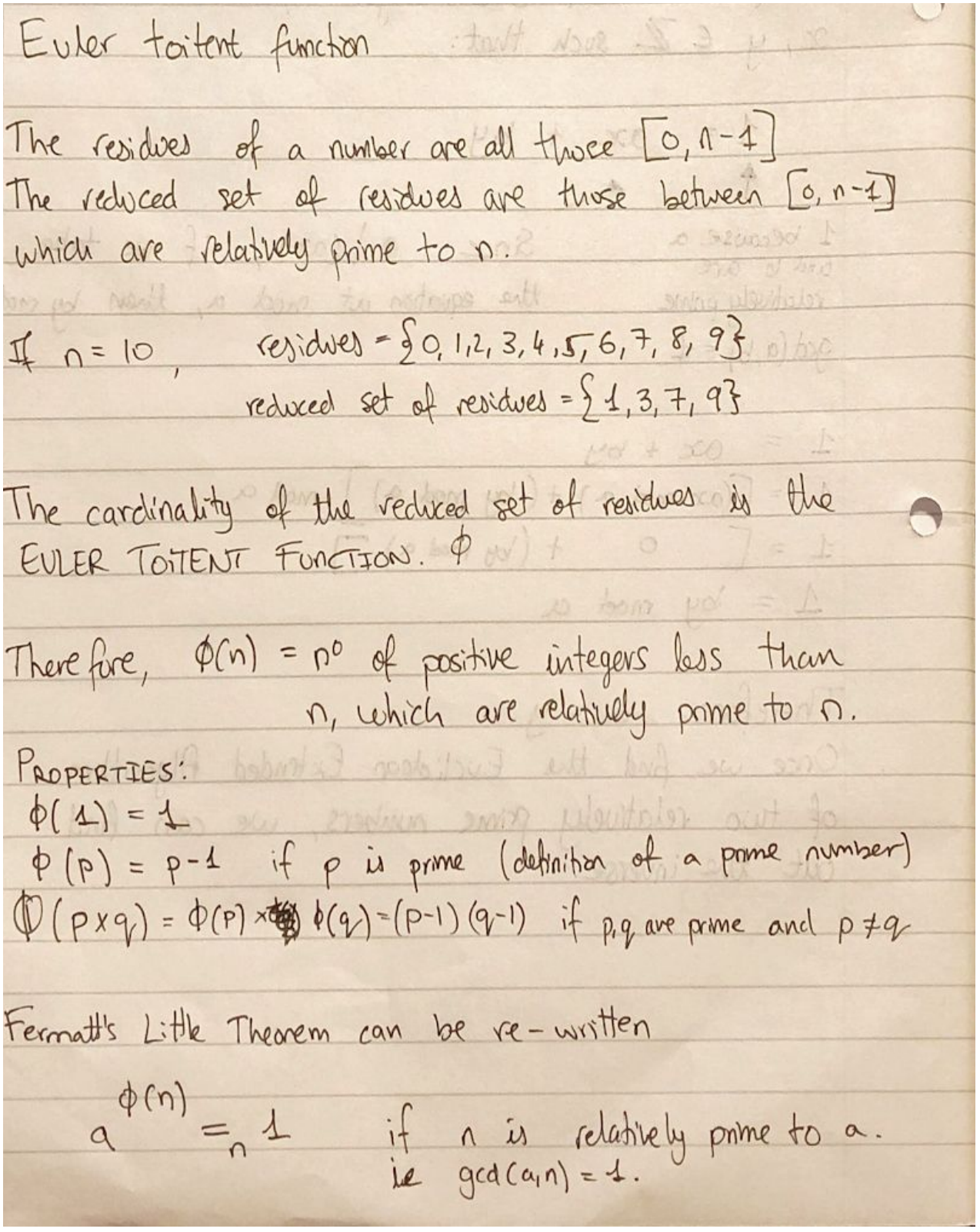
E.g. $4^6 \bmod 7 = 1$.

We can do $= (4^2 \times 4^2 \times 4^2) \bmod 7$

$= (16 \times 16 \times 16) \bmod 7$

$= (16 \% 7 \times 16 \% 7 \times 16 \% 7) \% 7$

$= (2 \times 2 \times 2) \% 7 = 1$.



EXAM: compute fi of some number

RSA Algorithm Properties

Is a block cipher, it encrypts blocks using public key cryptography. The plaintext and ciphertext are positive integers between 0 and n-1.

- A typical size for n is 1024 bits
- That is, *n* is less than 2^{1024}

The security of the algorithm depends on the key and we are going to use a very long key (more than 100 digit prime number).

We encrypt each block which has a binary value less than n. (that is less than or equal to $\log_2(n) + 1$. Each message sent is always smaller than n.

RSA encryption and decryption of M are defined as follows:

For some (properly chosen) values of e and d,

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

Our

public key PU = (e,n) and **private key** PR = (d,n)

Requirements for RSA

1. It is possible to find values of *e*, *d* such that M^{ed} *mod n* = *M* for all *M* < *n*
2. It is relatively easy to calculate M^e *mod n* and C^d *mod n* for all values of *M* < *n*
3. It is infeasible to determine *d* given *e* and *n*

Remember we can use extended euclid's algorithm to compute 1.

RSA Algorithm

Given that we know *p* and *q* we can easily compute *n* and *Φ(n)* but if we do not know both *p* and *q* and only knows *n* it becomes very difficult given that the numbers are very large. Calculating *Φ(n)* will be very difficult and therefore calculating *d* will also be very cumbersome.

TIP: when choosing e, it is a safe bet to choose the smallest prime number that does not divide phi(n), what we choose for e does not affect security.

- Generation of a public/private key pair:
  1. Generate $p = 47$, $q = 71$.
  2. Compute $n = p \times q = 3337$ and $\phi(n) = (p-1) \times (q-1) = 46 \times 70 = 3220$
  3. Choose $e = 79$ (randomly in the interval [1..3220])
  4. Compute $d = 79^{-1} \bmod 3220 = 1019$.
  5. Public key $(e, n) = (79, 3337)$, private key $(d, n) = (1019, 3337)$
- Encryption with key $(e, n) = (79, 3337)$:
  1. Break message M into blocks, e.g., 688 232 687 966 668 ...
  2. Compute $C_1 = 688^{79} \bmod 3337 = 1570$, $C_2 = ...$
- Decryption with key $(d, n) = (1019, 3337)$:
  1. Compute $M_1 = 1570^{1019} \bmod 3337 = 688$, $M_2 = ...$

Calculating Private Key, d

- We have $n = 187$, $\phi(n) = 160$, $e = 7$.
- d can be computed using the Extended Euclid algorithm

$$D = \gcd(A, B) = A \times x + B \times y$$

as follows:

- Since d is such that $e \times d =_{\phi(n)} 1$, we can compute

$$1 = \gcd(\phi(n), e) = \phi(n) \times x + e \times d$$

- It must be $1 < d < \phi(n)$, so when $y < 0$ we simply reason modulo $\phi(n)$.

- In this case:

$$1 = \gcd(160, 7) = 160 \times x + 7 \times y$$

A	B	$\lfloor A/B \rfloor$	D	x	y
160	7	22	1	−1	23
7	6	1	1	1	−1
6	1	6	1	0	1
1	0	–	1	1	0

That is, $1 = \gcd(160, 7) = 160 \times (-1) + 7 \times 23$. Check: $7 \times 23 =_{160} 1$.

So, we can pick $d = y = 23$.

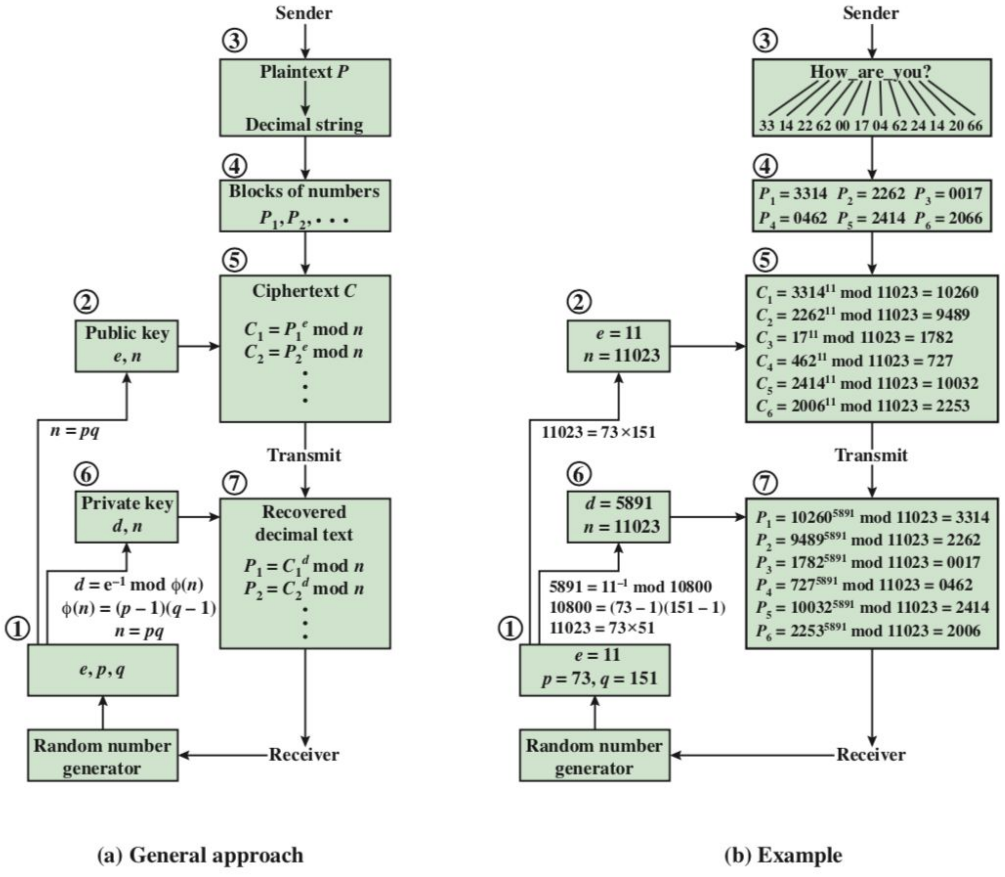
In the case that we get a y or a d that is NEGATIVE, d MUST be positive, in this case we must remember that it is the number modulo n. Therefore it will never be negative.

Simplifying Large Exponents

- To encrypt a plaintext input $M = 88$, we need to calculate $C = M^e \bmod n = 88^7 \bmod 187 = 11$.
- We can do this by exploiting properties of modular arithmetic:
  - $88^7 \bmod 187 = ((88^4 \bmod 187) \times (88^2 \bmod 187) \times (88^1 \bmod 187)) \bmod 187$
  - $88^1 \bmod 187 = 88$
  - $88^2 \bmod 187 = 7744 \bmod 187 = 77$
  - $88^4 \bmod 187 = 59,969,536 \bmod 187 = 132$
  - $88^7 \bmod 187 = (88 \times 77 \times 132) \bmod 187 = 894,432 \bmod 187 = 11$

General Look at RSA

- In this simple example, plaintext is an alphanumeric string.
- Each plaintext symbol is assigned a unique code of 2 decimal digits (e.g., a = 00, A = 26).
- A plaintext block consists of 4 decimal digits, or 2 alphanumeric characters.
- Circled numbers indicate order in which operations are performed.



Correctness of RSA

For RSA to be correct we must show that there exists an *e, d* and *n* such that:

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

For all $M < n$, we can show that the above holds if and only if *e* and *d* are multiplicative inverses:

$d =_{\Phi(n)} e^{-1}$ *if and only if*

$de =_{\Phi(n)} 1$ *if and only if*

$de \bmod \Phi(n) = 1$

But this is only true if *d (and e)* is relatively prime to *Φ(n)*

RSA Security

The security of RSA depends on the:

- **Computation of our secret key d given (e,n)**: which is as difficult as factorization. If we can factor *n = pq*, then we can compute *Φ(n)* and therefore *d*. *There is no polynomial time algorithm known that can do this if n is at least 1024 bits.*
- **Computation of M_i given C_i and *(e,n)*:** there is no proof that we actually need *d* to recover the plaintext for the ciphertext. We do not have an efficient way of doing it but we do not have a proof.

Hence, progress in number theory or the invention of a quantum computers could break RSA.

RSA Worked Example Public and Private Key Generation

Summary:
Given two primes, *p* and *q*

1. Calculate *n* and *Φ(n)*
2. Find an encryption key *e* (public key), such that *e* is relatively prime to *Φ(n)*
   Ie. that gcd(Φ(n),e) = 1
3. Calculate the inverse of *e* by applying the proof for Theorem 1 presented earlier. Then using Euclid's Extended Algorithm we calculate *y* in *1 = ax + by* where *y = c* or in this specific example *c* will be our decryption key *d* and *b* our encryption key *e*

Asymmetric Algorithms for Secret Key Distribution

When we want to use symmetric cryptosystems we need so many keys that we have to distribute, with PKE this number is much smaller and manageable. Therefore, we can use **PKE algorithms** to support (faster) symmetric cryptography. We will look at doing so with RSA and with the Diffie-Hellman Key Exchange.

Secret Key Distribution with RSA

1. Alice can create a symmetric key *K*, and encrypt a message $M \rightarrow E_k(M)$
2. She can then encrypt the key (which is much shorter than the message and thus more efficient) with Bob's Public Key $K_B \rightarrow K^{KB} \bmod n$.
3. Alice can now send ($K^{KB} \bmod n$, $E_k(M)$)
4. Bob can then decrypt the key and use it and decrypt the message sent, moreover they can now use that key for the rest of the communication.

Used for SSL: Alice choses a secret, encrypts it with Bob's public key and the rest of the session communication uses the secret chosen by Alice.

If Bob's private key gets compromised, then K can be recovered and an intruder can read any previous traffic. A way to fix this which does not rely on the private key of an individual is the **Diffie-Hellman key exchange.**

Diffie-Hellman Key Exchange

It is very easy to mix two colours but once they are mixed, it's very difficult to separate them to the original colours again. The following occurs, there is a public colour, Alice and Bob choose a colour. Each of them mixes the public colour with their colour and then send that outcome to each other, then they mix it again with their colour again.

The diffie-hellman key exchange is a means by which a same shared secret key can be shared between two parties in an insecure communication facility. It is based on the discrete logarithm problem. The protocol is **secure** only if the **authenticity** of the **two** participants can be established.

Discrete Logarithms

A **primitive root** *s* of a prime number *p* is a number whose powers generate 1, … , p-1. Similarly to the proof of fermat's little theorem, this means that there is a sequence of numbers *1* t*o p-1.* For any integer *b* and a primitive root *s* of prime number *p*, we can find a unique exponent *i* such that

$$\forall b \in \mathbb{Z}.\, \exists i \in \{0, \ldots, p-1\}.\; b = s^i \bmod p$$

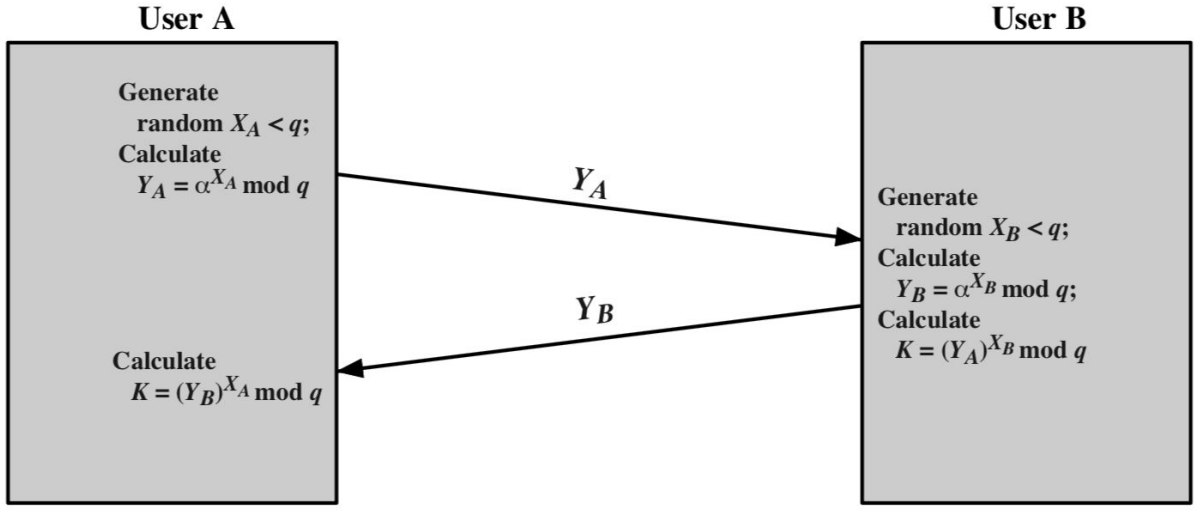
Where *i* is the **discrete logarithm** of *b* for base *s*, *mod p*
Computing s^i *mod p* is easy but if I give you *b* and ask for the *i* that is tricky, even if I give you *s* and *p.*

Example primitive roots

- 2 is a primitive root of 5 since
  - $2^1 \bmod 5 = 2 \bmod 5 = 2$
  - $2^2 \bmod 5 = 4 \bmod 5 = 4$
  - $2^3 \bmod 5 = 8 \bmod 5 = 3$
  - $2^4 \bmod 5 = 16 \bmod 5 = 1$
- 3 is a primitive root of 5 since
  - $3^1 \bmod 5 = 3 \bmod 5 = 3$
  - $3^2 \bmod 5 = 9 \bmod 5 = 4$
  - $3^3 \bmod 5 = 27 \bmod 5 = 2$
  - $3^4 \bmod 5 = 81 \bmod 5 = 1$

1. Principals share a prime number q and an integer α that is a primitive root of q. Both q and α may be public, or A could send them in the first message.
2. A and B generate random numbers X_A and X_B (respectively) both less than q.
   X_A and X_B are the **private keys**.
3. A computes $Y_A = \alpha^{X_A} \bmod q$, B computes $Y_B = \alpha^{X_B} \bmod q$.
   Y_A and Y_B are the **public keys** (a.k.a. "Diffie-Hellman half keys").
4. A and B exchange Y_A and Y_B.
5. A computes $K_A = Y_B^{X_A} \bmod q$, B computes $K_B = Y_A^{X_B} \bmod q$.
   Keys are equal, i.e., $K_A = K_B$:

$$\begin{aligned} K_A &= Y_B^{X_A} \bmod q \\ &= (\alpha^{X_B} \bmod q)^{X_A} \bmod q = (\alpha^{X_B})^{X_A} \bmod q \\ &= \alpha^{X_A X_B} \bmod q = (\alpha^{X_A})^{X_B} \bmod q \\ &= (\alpha^{X_A} \bmod q)^{X_B} \bmod q = Y_A^{X_B} \bmod q \;=\; K_B \end{aligned}$$



They both send everything in plaintext, and they never send K but they send Y_A and Y_B. Even if you know q and alpha, it's very difficult to compute the exponents because of the **discrete logarithm problem.**

- Keys are **unauthenticated** and the Diffie-Hellman key exchange is vulnerable to a **Man-in-the-middle Attack**

+ The shared key is never transmitted in any way.
+ An adversary can only work with q, α, Y_A and Y_B, because X_A and X_B are private. Therefore, he will have to take a discrete logarithm to calculate the key, for example:

$$X_B = \mathrm{dlog}_{\alpha,q}(Y_B)$$

  To calculate Bob's private key.

The security relies on how easy it's to calculate exponents modulo prime, but how tard it is to calculate discrete logarithms (infeasible for large primes).

Security depends on the difficulty of computing discrete logarithms.

Diffie-Hellman Calculating the Secret Key Example

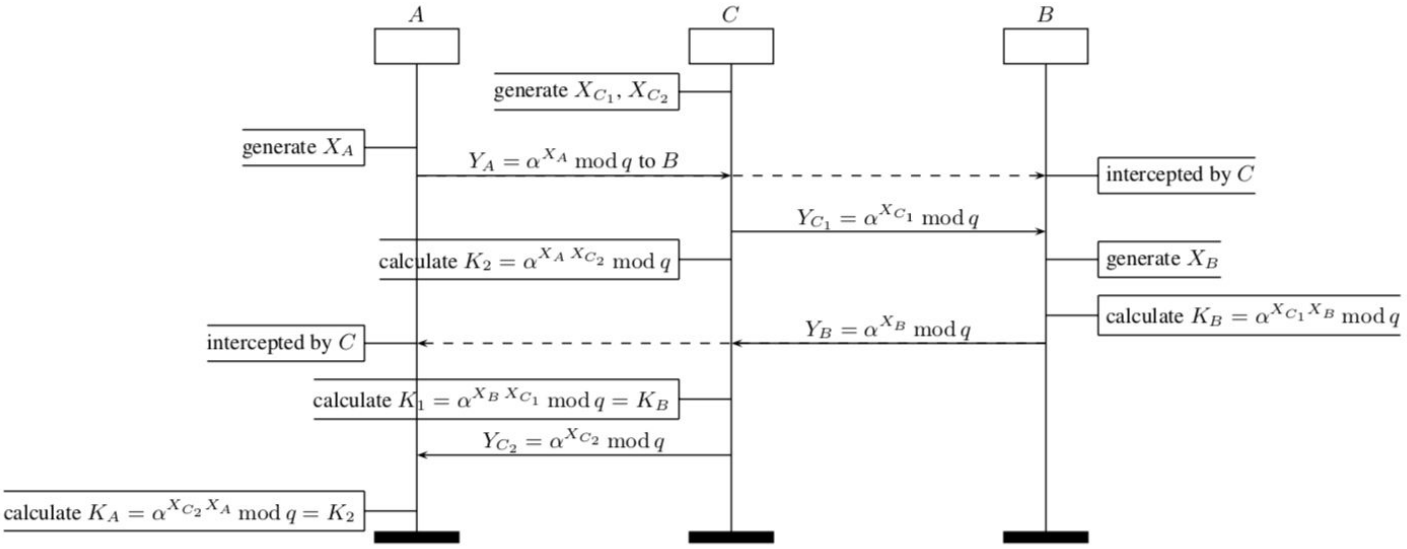
- A and B choose prime number $q = 353$ and $\alpha = 3$ (which is one of the primitive roots of 353).
- A and B select private keys $X_A = 97$ and $X_B = 233$.
- Each computes its public key:
  - A computes $Y_A = \alpha^{X_A} \bmod q = 3^{97} \bmod 353 = 40$.
  - B computes $Y_B = \alpha^{X_B} \bmod q = 3^{233} \bmod 353 = 248$.
- After they exchange public keys, each can compute the common secret key K:
  - A computes $K = (Y_B)^{X_A} \bmod 353 = 248^{97} \bmod 353 = 160$.
  - B computes $K = (Y_A)^{X_B} \bmod 353 = 40^{233} \bmod 353 = 160$.
- Now they case use the symmetric key K to encrypt the messages they want to exchange.

Diffie-Hellman Attacking the Secret Key Example

- Attacker C knows: $q = 353$, $\alpha = 3$, $Y_A = 40$ and $Y_B = 248$.
  - In this simple example, it would be possible by brute force to determine the secret key $K = 160$.
  - In particular, C can determine K by discovering a solution to
    - the equation $3^a \bmod 353 = 40$ or
    - the equation $3^b \bmod 353 = 248$.
  - Brute-force approach: calculate powers of 3 mod 353, stopping when the result equals either 40 or 248.
    - Desired answer is reached with the exponent value of 97, which provides $3^{97} \bmod 353 = 40$.
- With larger numbers, the problem becomes impractical.

Diffie-Hellman Man-in-the-middle Attack

Bob does not know that what Alice sent was actually sent by Alice and same thing for Bob.



Now *A* and *B* think that they share a secret key, but instead *A* shares secret key $K_A = K_2$ with *C* and *B* shares secret key $K_B = K_1$ with *C*.

We must authenticate our senders to ensure we are getting out keys from the right person.

- All future communication between Bob and Alice is compromised in the following way.
  1. *A* sends an encrypted message *M*, i.e., $E(K_2, M)$.
  2. *C* intercepts the encrypted message and decrypts it to recover *M*.
  3. *C* sends to Bob either
     - $E(K_1, M)$, if *C* simply wants to eavesdrop on the communication without altering it, or
     - $E(K_1, M')$, where M' is any message, if *C* wants to modify the message going to *B*.

We can overcome this vulnerability with the use of **digital signatures and public-key certificates** to achieve mutual authentication for A and B.

Diffie-Hellman for Three or more parties

Given a Diffie-Hellman group (α, q), three honest parties Alice, Bob and Carol can generate together a secret key $K = \alpha^{X_A X_B X_C} \bmod q$ by:

We would just make sure we send all the half/third/quarter… keys to everyone to reach consensus on a key.

El Gammal variant of Diffie-Hellman Key Exchange

This is similar to Diffie-Hellman but Bob sends a message encrypted as well with the key K.
El Gamal = diffie-hellman + message exchange

- Setup: same as Diffie-Hellman. Public prime q and generator α.
- Moreover, let E be any symmetric encryption function.
- Schema
  1. A chooses X_A and computes $Y_A = \alpha^{X_A} \bmod q$.
     $$A \to B\text{: } Y_A.$$
  2. B chooses integer X_B, computes $Y_B = \alpha^{X_B} \bmod q$, and computes key $K = Y_A^{X_B} \bmod q$.
     $$B \to A\text{: } (E(M, K), Y_B)$$
  3. A computes $K = Y_B^{X_A} \bmod q$ and uses K to decrypt $E(M, K)$.

  Red parts are Diffie-Hellman.

Massey-Omura Scheme

Encryption without shared keys based on the **discrete logarithm problem**.

- The agents share public number prime p
- They have a public and private key

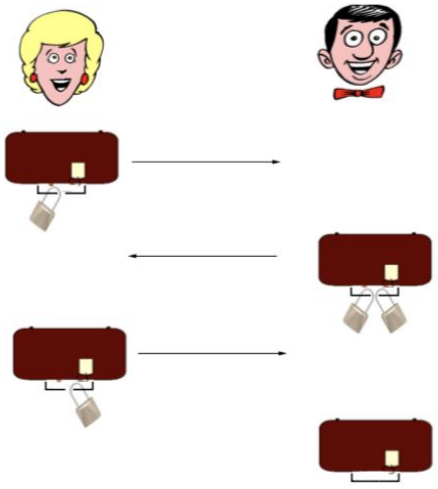
1. Alice encrypts and sends to Bob
2. Bob encrypts and sends to Alice
3. Alice decrypts and sends to Bob
4. Bob decrypts and can read the message

Step 1 $A \to B$: $m^{e_A} \bmod p$

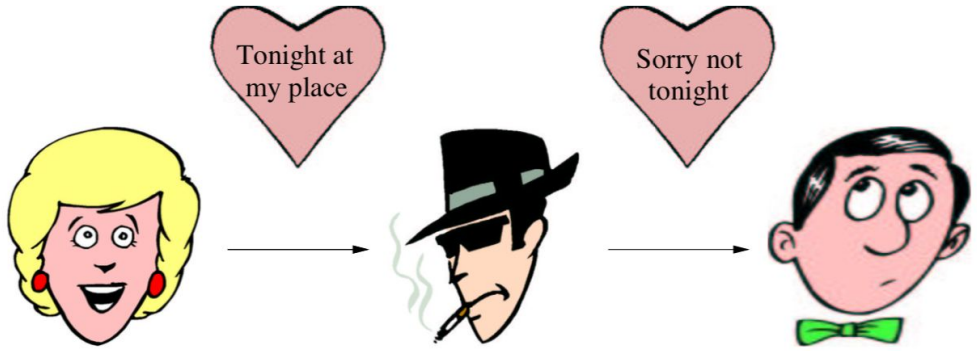
Step 2 $B \to A$: $m^{e_A e_B} \bmod p$

Step 3 $A \to B$: $m^{e_A e_B d_A} \bmod p \quad (= m^{e_B})$

Step 4 $m^{e_A e_B d_A d_B} \bmod p \quad (= m)$



Message Integrity and Cryptographic Hashes



Message integrity: the property that data has not been altered in an unauthorized manner. A way to ensure integrity is to use hashes.

We can use a hash to create a fingerprint of the data.

Hash function *h(x)* has the properties:

- **Compression:** takes input of arbitrary length and outputs and output of fixed length n.
- **Polynomial time computable**

An example would be when we check if a DES is a valid key to see if it is odd, we are doing the sum of all the bits in a byte modulo 2, we take something or arbitrary length and we output something of fixed length (1).

Cryptographic hash functions *h(x)* are hash functions with additional properties:

- **Compression:** takes input of arbitrary length and outputs and output of fixed length n.
- **Polynomial time computable**
- **One-way (preimage resistant):** given output, it very difficult to invert it.

Either has:

- **2nd-preimage resistance:** it is computationally infeasible to find a second input that is different to the first and has the same output as the first.

  It is impossible that given x to find an $x' \neq x$ such that $h(x) = h(x')$.

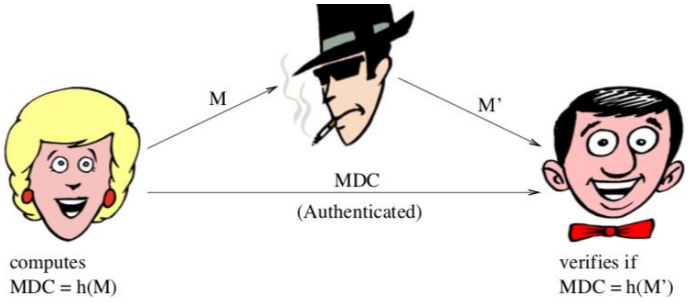
  It is unlikely that two different inputs will create the same hash.

or

- **Collision resistance:** it is difficult to find two distinct inputs that create the same hash.

  It is impossible that two different inputs will create the same hash.

Hash value also called **message digest** or **modification detection code**.



Requires 2nd-preimage resistance and authenticated MDC.
Typical application: signed hashes.

We can now apply message integrity and send the message along with a an authenticated **Modification Detection Code** (which is a hash of the message) in an authenticated way. In this way Bob will be able to verify that the message's hash is the same as the MDC sent.

We can carry out a Birthday Attack
Imagine Alice and Bob have a contract and they have agreed on a price. But Bob is malicious and is going to change the price to much higher to bankrupt it.

1. B can then create version x of the contract (the good version)
2. and version y (the bad version).
3. Bob generates all the versions of the good and bad messages until he finds a $h(x_i) = h(y_j)$
   This can be done by adding small variations to the message which mean the same but have different characters, like so:

   { This letter is / I am writing } to introduce { you to / to you } { Mr. / } Alfred { P. / }

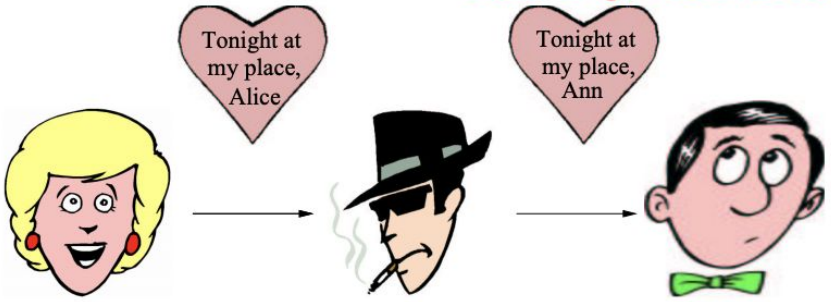
4. A then signs $h(x_i)$ *and B uses that signature for* y_j. Because whatever she signs for x_i will also be valid for y_j since the output of the hash is the same.

On average the work required for 2^{62} bits of a message is 2^{32}. To ensure **collision avoidance** you must double your hash size.

Message Authentication

Although a message may not be tampered with, it may not come from the correct source.

- Message authentication also called data-origin authentication.



Observe that message authentication implies message integrity.

- Contrast to entity (or user) authentication

We must differentiate between:

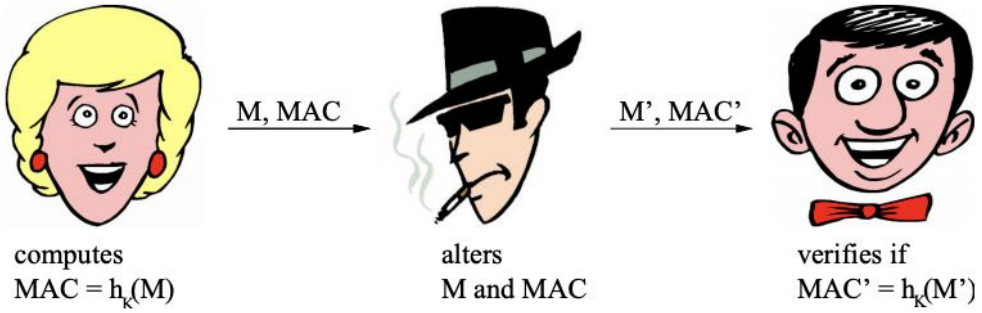
- **Entity (user) authentication:** you want to make sure who the person sending you the message says who he/she claims he/she is.
- **Data-origin (message) authentication:** you want to know who has sent the message (the message might be for you or it might be a broadcast).

Luca may digitally sign a communication to all the class but that doesn't necessarily mean that the Luca and I have been sending messages back and forward.

To ensure that our message comes from the right source we must use **Message Authentication Codes (MACs)** and **digital signatures**.

Message Authentication Codes (MACs)

A **MAC Algorithm** is a family of hash functions h_k each given by the specific secret key has it been used. The hk must be **computation-resistant** meaning that given zero or more MAC pairs $(x_i , h_k (x_i))$, it is infeasible to compute $(x, h_k (x))$ for any new input $x \neq x_i$.



As seen above, alice and bob don't share an MDC, in other words they do not share a hash of the message rather a hash with a symmetric key that they both hold of a message. In this example **charlie** may still fool Bob by changing both the message and the MAC but he must have h_k.

With this scheme we **achieve message authentication** for verified messaged but we **cannot** achieve **non-repudiation** because Bob could have forged everything since he knows all the ingredients, to add non-repudiation we must use public key cryptography. **Freshness** (the fact that the message was created now) **is not achieved**.

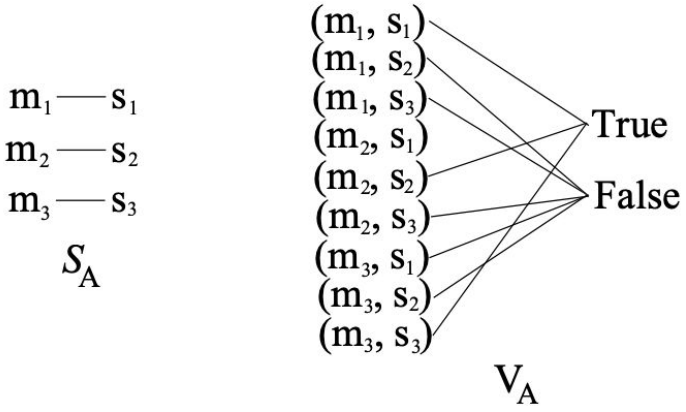
Digital Signatures

We must produce a proof of data origin, we must prove that the message originated from a particular person. Public key cryptography supports **message authentication** (in the same way that MACs do that with symmetric keys) and **nonrepudiation** (because of the fact that each person has his own key).

For a signature we have:

- $\mathcal{M}$ is set of messages that can be signed.
- $\mathcal{S}$ is set of elements called signatures, e.g., n-bit strings.
- $S_A : \mathcal{M} \rightarrow \mathcal{S}$, is a signing transformation for entity A, and kept secret by A.
- $V_A : \mathcal{M} \times \mathcal{S} \rightarrow \{true, false\}$ is a verification transformation for A's signature and is publicly known.

Example



- Signing procedure: A creates a signature for $m \in \mathcal{M}$ by:
  Compute $s = S_A(m)$ and transmit pair (m, s).
- Verification procedure. B verifies A's signature of (m, s) by:
  Compute $u = V_A(m, s)$. Accept signature only if $u = true$.
- Secrecy requires: it is hard for any entity other than A to find, for any $m \in \mathcal{M}$, an $s \in \mathcal{S}$, where $V_A(m, s) = true$.

  Or in other words, it must be difficult to find a signature that matches with a message if you're not A.

If you also want **confidentiality**, you will have to encrypt with the public key of bob.

Implementing Digital Signatures

Digital Signing can be done with reversible* public-key encryption systems.

*meaning that we can first encrypt and then decrypt or vice-versa.

$$D_d(E_e(m)) = E_e(D_d(m)) = m \qquad \text{for all } m \in \mathcal{M}.$$

The Digital Signature Schema is:

1. Let M and C be message and signature space with $M = C$
2. Let e be your public key and d be your private key
3. Define the signing function S_A to be D_d (decryption [remember it's just a function] with our private key)
4. Define V_A by

$$V_A(m, s) = \begin{cases} true, & \text{if } E_e(s) = m, \\ false, & \text{otherwise} \end{cases}$$

Since our keys are very long, we mustn't use the actual message, rather using a **hash** of the message is more computationally efficient.

If we do not hash, not only we are making it computationally complex but the schema admits a **forgery attack:**

1. Attacker B selects a random s in S and compute $m = E_e(s)$ with Alice's public key
2. Since the message and signature space is the same S = M, he can submit (m,s) as a message signature
3. Since the encryption is reversible (same as decryption) and we have done it with a Alice's public key, the verification returns *true* even though A didn't sign m

The solution is to define a subset of the message space which are the signable messages ($M' \subset M$) and redefine our verification function to only accept these.

Redefine verifier $V_A : \mathcal{S} \to \{true, false\}$ as:

$$V_A(s) = \begin{cases} true, & \text{if } E_e(s) \in \mathcal{M}' \\ false, & \text{otherwise} \end{cases}$$

Messages can be recovered since $m = E_e(s)$.

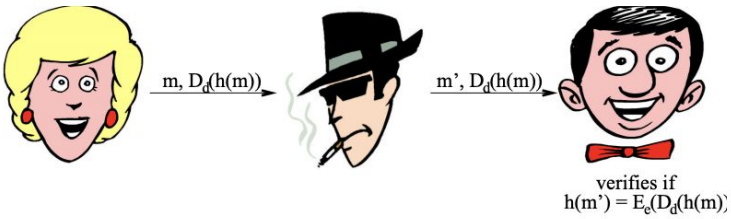
Secure when $\mathcal{M}'$ is a sufficiently small subset of $\mathcal{M}$.

The way of creating this subset is by using a hash function.

- RSA provides a realization of digital signatures:

  $$D_d(E_e(m)) = E_e(D_d(m)) = m$$

  Forgery prevented by signing messages with fixed structure, e.g.,
  - Message names its sender, or (more typically).
  - Cryptographic hash signed, sent with the message.



  Pair can additionally be encrypted for confidentiality.
- Also: **Digital Signature Algorithm (DSA)**, a Federal Information Processing Standard for digital signatures. It was proposed by the National Institute of Standards and Technology (NIST) in August 1991 for use in their **Digital Signature Standard (DSS)**.

- Symmetric/asymmetric cryptography can be used to achieve confidentiality, integrity and authentication assuming properly distributed keys.
- Asymmetric cryptography simplifies key distribution and to digitally sign messages (message authentication + non-repudiation.

Security Protocols

How to we make sure that we have entity authentication if Alice is a customer and Bob is the bank? We need to establish security protocols that provide the security properties necessary in each situation.

Solutions involve protocols like IPSEC, KERBEROS, SSH, SSL/TLS, SET, PGP

Protocol: consists of a set of rules (conventions) that determine the exchange of messages between two or more principals. *Ie a distributed algorithm to communicate.*

Tip to remember, if you meet the queen there is a set of rules that tell you how to interact.

Security protocol: Security (or cryptographic) protocols use cryptographic mechanisms to achieve security objectives (Entity or message authentication, key establishment, integrity, timeliness, fair exchange, non-repudiation)

Messages, communication, protocols

We are going to have:
Names: A, B or Alice and Bob
Keys: *K* and **inverse keys** K^{-1}

- Encryption: $\{M\}_K$
  - Encryption with A's key: $\{M\}_{KA}$
- Signing: $\{M\}_{K-1}$
  - Signing with A's key: $\{M\}_{KA-1}$

Symmetric keys: $\{M\}_{KAB}$
Nonces: N_A, N_1, … which are fresh data items used for challenge/response

- subscript used but not because someone can find out who it was generated by, it is just a number, to verify where it comes from we must do that, verify it)

Timestamps: *T* denote time (e.g. for key expiration)
Message Concatenation: M_1,M_2

Example: $\{A, T_1, K_{AB}\}_{KB}$ → Alice's Name, Timestamp and a symmetric key encrypted with Bob's public key.

Communication is specified in something called "Alice and Bob Notation"

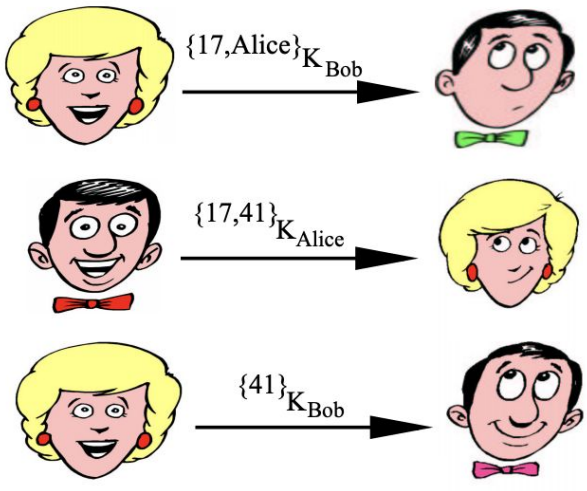
$$A \rightarrow B : \{A, T_1, K_{AB}\}_{K_B}$$

Sender → Receiver : What is sent.

An Authentication Protocol: The Needham-Schroeder Public Key protocol (NSPK)

1. Alice sends a nonce, pairing it with her name, encrypting it with the public key of Bob and sending it to Bob.
2. Bob reads that and sends the nonce back encrypted with the public key of Alice.
3. Alice reads that and send back the nonce Bob created encrypted with Bob's public key

$$\begin{aligned} 1.\quad & A \rightarrow B : \quad \{NA, A\}_{K_B} \\ 2.\quad & B \rightarrow A : \quad \{NA, NB\}_{K_A} \\ 3.\quad & A \rightarrow B : \quad \{NB\}_{K_B} \end{aligned}$$



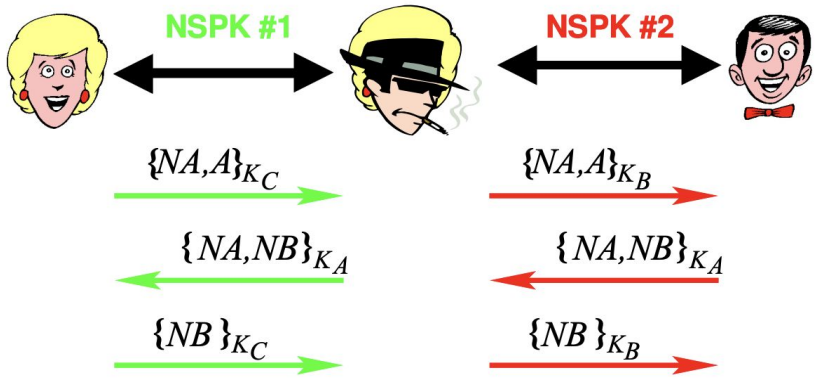
Our protocol should:

- Message authentication
- Ensure timeliness (freshness)
- Guarantee secrecy of some items (like generated keys)

NSPK: Man-in-the-Middle Attack

We are going to assume that the attacker is **not able to break cryptography**. He may be **passive** and overhear all communications or **active** and can intercept and generate messages.

In our attack, Charlie is going to play two games, one with Alice and one with Bob. ***Just like Luca plays chess grandmaster.***



$$X \to Y : \{N1, X\}_{K_Y}$$
$$Y \to X : \{N1, N2\}_{K_X}$$
$$X \to Y : \{N2\}_{K_Y}$$

B believes he is speaking with *A*!

Charlie simply mirrors Alice's communication with him with Bob and Bob does as if he was speaking to Alice. Bob thinks he is speaking with Alice but really he's speaking with Charlie.

The problem is that although Alice identifies herself at first, Bob does not in his first message (step 2), therefore Alice has no certification that Bob is Bob, rather anyone with her public key could send her back a response and she would believe it if the nonce she sent was true.

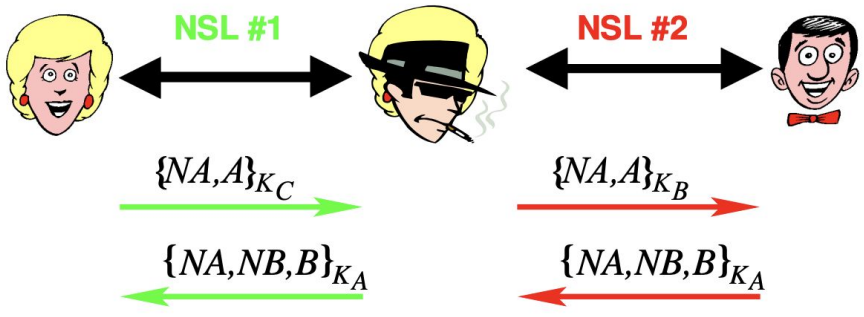
- Problem in step 2:

  $$B \to A : \{NA, NB\}_{K_A}$$

  Agent *B* should also give his name: $\{NA, NB, B\}_{K_A}$.

- The improved version is called NSL protocol.

NSL Protocol



$$X \to Y : \{N1, X\}_{K_Y}$$
$$Y \to X : \{N1, N2, Y\}_{K_X}$$
$$X \to Y : \{N2\}_{K_Y}$$

A aborts the protocol execution!
(or ignores the message)

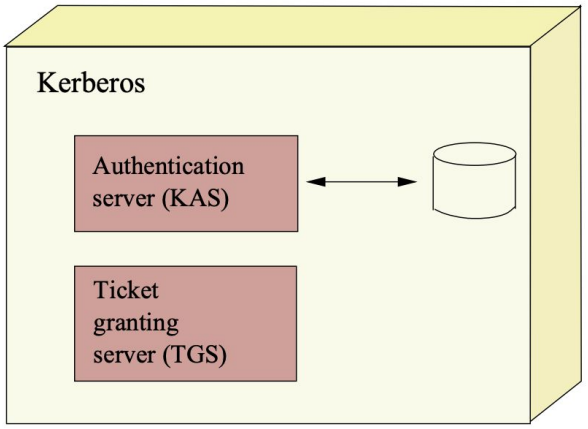
Now this protocol is safe to a man-in-the-middle attack.

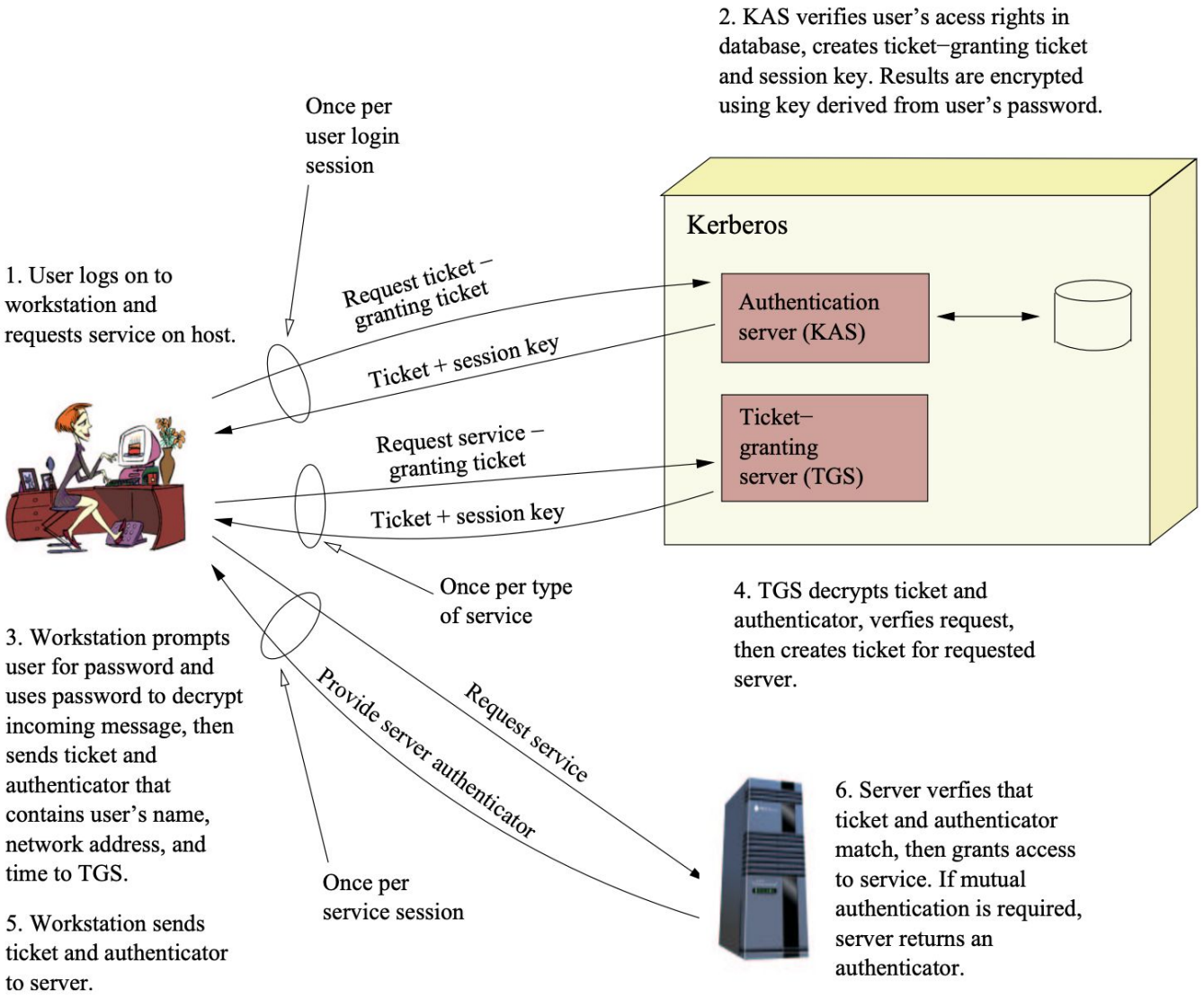
Kerberos

Kerberos is an authentication, access control and auditing system.

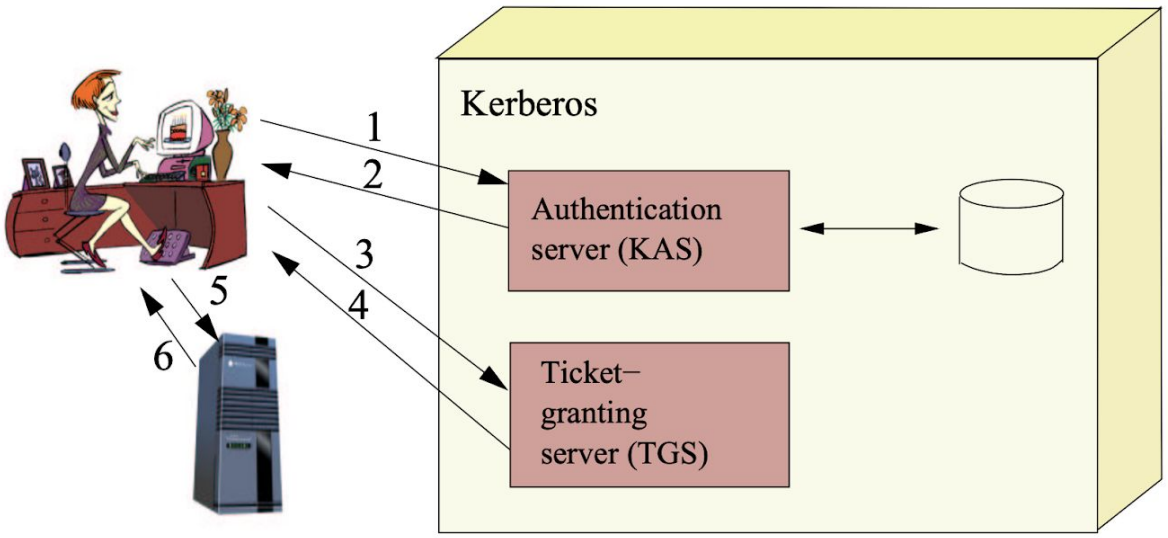
It works by having a system that provides two servers:

- **authentication server (KAS)**
- **ticket granting server (TGS)** that authorises or controls access to resources.





Like logging on to the king's system, that session lasts for a few hours. If we want to print we must log in into follow me printing, namely we have to authenticate ourselves at the printer ie we are getting a ticket to access the printing services.



Authentication	messages 1 and 2.	Once per user login session.
Authorization	messages 3 and 4.	Once per type of service.
Service	messages 5 and 6.	Once per service session.

KERBEROS Authentication Phase

1. Alice sends a message to the KAS saying “I’m Alice, please give me access to the TGS”
2. KAS sends to Alice a message saying “Here is your key to talk with that server, here is the name of the server, here is the timestamp and here is your authentication ticket” all encrypted with a symmetric key between Alice and the server (typically generated from Alice’s password).

$$1.\ A \rightarrow KAS: \quad A, TGS$$
$$2.\ KAS \rightarrow A: \quad \{K_{A,TGS}, TGS, T_1, \underbrace{\{A, TGS, K_{A,TGS}, T_1\}K_{KAS,TGS}}_{AuthTicket}\}K_{AS}$$

- The auth ticket is encrypted with a key between KAS and TGS so Alice cannot decrypt it.
- $K_{A,TGS}$ *is a symmetric key for alice to use with the TGS, it has a lifetime of several hours*
- K_{AS} *derived from the user’s password*

The idea of timeness is introduces, first you have the time of the session, then the time of the ticket which is less.

KERBEROS Authorization Phase

Assuming Alice wants to access resource B.
3. Alice sends the authentication ticket along an authenticator (which has seconds lifetime) to TGS
4. The TGS sees that he receives to messages, one which he cannot decrypt and another which he can decrypt. He decrypts it (the AuthTicket), in there he finds the key to decrypt the authenticator. He then authenticates alice and gives her a service ticket to the resource.

$$3.\ A \rightarrow TGS: \underbrace{\{A,\ TGS,\ K_{A,TGS},\ \mathcal{T}_1\}_{K_{KAS,TGS}}}_{AuthTicket}, \underbrace{\{A,\ \mathcal{T}_2\}_{K_{A,TGS}}}_{authenticator}, B$$

$$4.\ TGS \rightarrow A: \{K_{AB}, B, \mathcal{T}_3, \underbrace{\{A,\ B,\ K_{AB},\ \mathcal{T}_3\}_{K_{B,TGS}}}_{ServTicket}\}_{K_{A,TGS}}$$

The role of the **authenticator** is to have a short validity to prevent replay attacks and to check that it is actually alice that really has requested this service.

KERBEROS Service Phase

5. Alice presents the service ticket and an authenticator
6. B will do the same as TGS, decrypt what he can and then use the key found in there to decrypt the authenticator, authenticate Alice and grant her access to the service for some time.

$$5.\ A \rightarrow B: \underbrace{\{A,\ B,\ K_{AB},\ \mathcal{T}_3\}_{K_{B,TGS}}}_{ServTicket}, \underbrace{\{A, \mathcal{T}_4\}_{K_{AB}}}_{authenticator}$$

$$6.\ B \rightarrow A: \{\mathcal{T}_4 + 1\}_{K_{AB}}$$

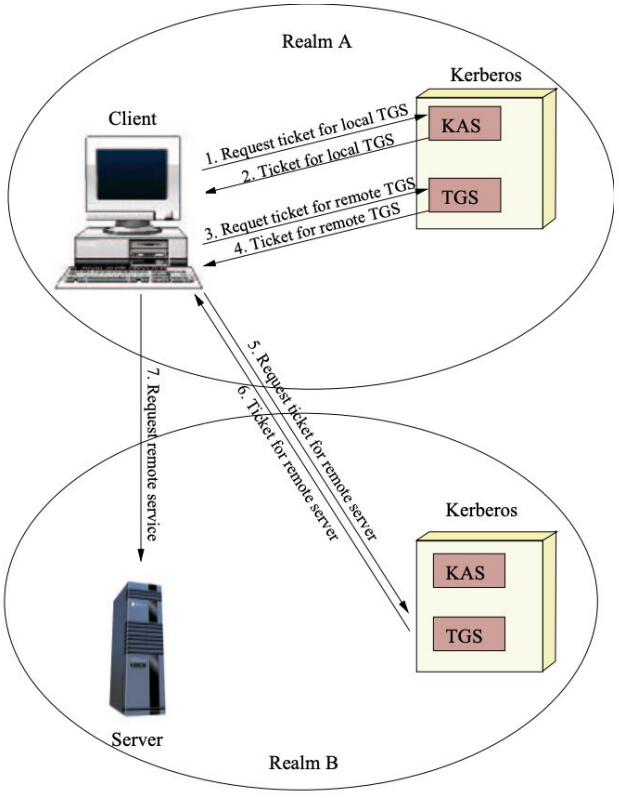
You first get a general ticket and then you can ask for tickets to other services like printing, or accessing student records until your main ticket expires.

Scalability of Kerberos

A **realm** is defined by a Kerberos server. A server stores user andd application server passowrds for the real,. Kerberos supports inter-realm protocols allowing a user to access a service in some other realm.

We can add a service that gives us a ticket to another Kerberos server. Just like we can use eduroam elsewhere.

You can think of it as passports, each country is the king of their own realm, meaning they can specify what you can do with your passports, but you can use your passport to go to other realms (countries) since everyone is using the same kerberos system.



To extend this service, we just need two additional steps. The only problem is that for many realms you need to distribute a lot of keys [$O(n^2)$ keys].

To request accessing another kerberos you would:

1. Authenticate yourself in your Kerberos

2. Ask for a ticket for another realm. If the two reals have exchanged keys and have compatible symmetric keys, the ticket that you will get will be able to talk to the other kerberos.
3. With the ticket for another realm you can access the service in the other kerberos.

The only problem is that the **kerberi must have agreed on keys beforehand.** *Much in the same way that you can only enter another country if they have recognised your passport.*

- Kerberos suffers from DDOS attacks
- The double encryption used in version 4 is redundant

EXAM: answer questions about kerberos or reason about NSPK protocols.

Passwords

Methods of authentication are often characterised as:

- Something you have
- Something you know
- Something you are

We can use cryptanalysis to break a cryptographic algorithm and learn a secret key.

Strong Authentication

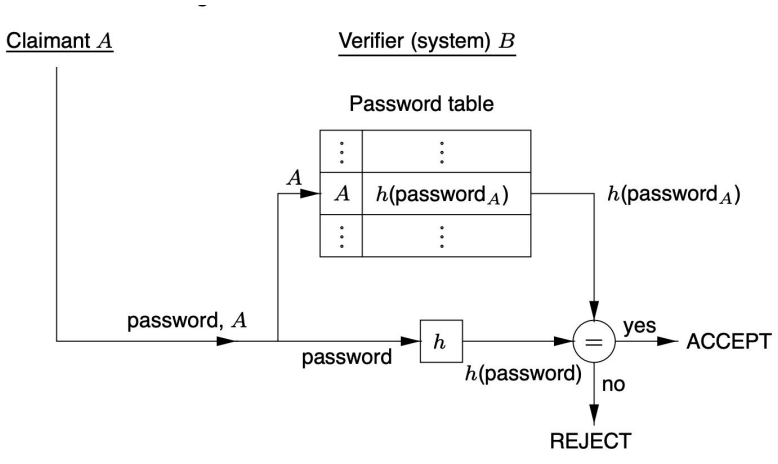
Can be achieved using **cryptographic challenge-response protocols:** one entity (the claimant) “proves” its identity to another entity (the verifier) by demonstrating knowledge of a secret known to be associated with that entity, without revealing the secret itself to the verifier during the protocol.

In other words, you authenticate yourself by using your private key and answering a challenge. The challenge may be a nonce a timestamp or have other forms.

But you might not want to reveal the secret itself, ie we do not want to reveal our private key.

Weak Authentication

Typically achieved by using passwords. The user enters a pair (userid, password) and the system verifies that it matches the corresponding data that it has in some database about this person.



Typically the password is stored hashed with some salt.

It is stored hashed with salt so that if we leak the information, the attacker is only able to recover the hashed password only.

Password rules:

- Change it often
- The longer, the better
- Be mysterious about it
- Should be changed often

Using weak passwords allows an attacker to carry out a **Dictionary Attack** in which the attacker tries every single word in the dictionary until he finds the correct password. To help with this we came up with rules, alphanumeric, lower case, upper case etc. We might also add password **ageing** (users are forced to change it periodically) this may make people use easy passwords or write them down if asked very frequently.

We must aim to have passwords which have a high **entropy** which refers to the uncertainty of the password and make it less vulnerable to exhaustive search or dictionary attacks. Ideally we would like to have passwords which are equiprobable, this would maximise the entropy of the passwords. The concept of being lost in the crowd providing anonymity is the best approach here.

- **PINs** can also be used but these are short, therefore these are most times used in conjunction with something else, like a password.

Multi-factor Authentication (MFA)

Multi-factor authentication (MFA) is an authentication method in which a user has to successfully present authentication factors from at least two of the three categories:

- **knowledge factors** ("things only the user knows"), such as passwords,
- **possession factors** ("things only the user has"), such as ATM cards or smartphones,
- **inherence factors** ("things only the user is"), such as biometrics. Requiring more than one independent factor increases the difficulty of providing false credentials.

One-time Passwords

Passwords that are used only once. This is secure because they are used only once, therefore if someone eavesdrops it is not usable for the next time you communicate with the system.

- We might have a **shared list** of one-time passwords (used OTP that we cross out)
- One-time passwords which are **updated, derived** from something (a number and update it all the time)
- One-time password sequences **based on a one-way function**

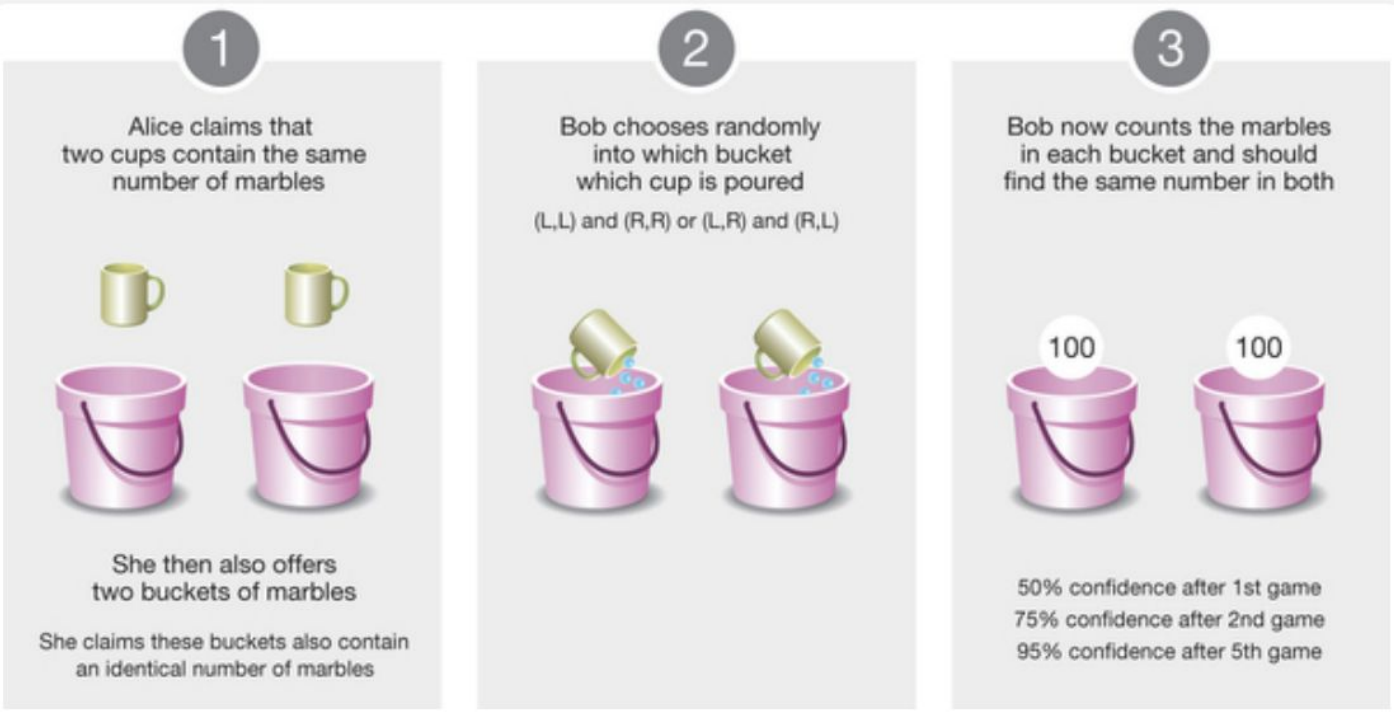
Zero-Knowledge Protocols

A way to protect passwords is to use zero-knowledge protocols. Namely using protocols that can authenticate you without knowing your password.

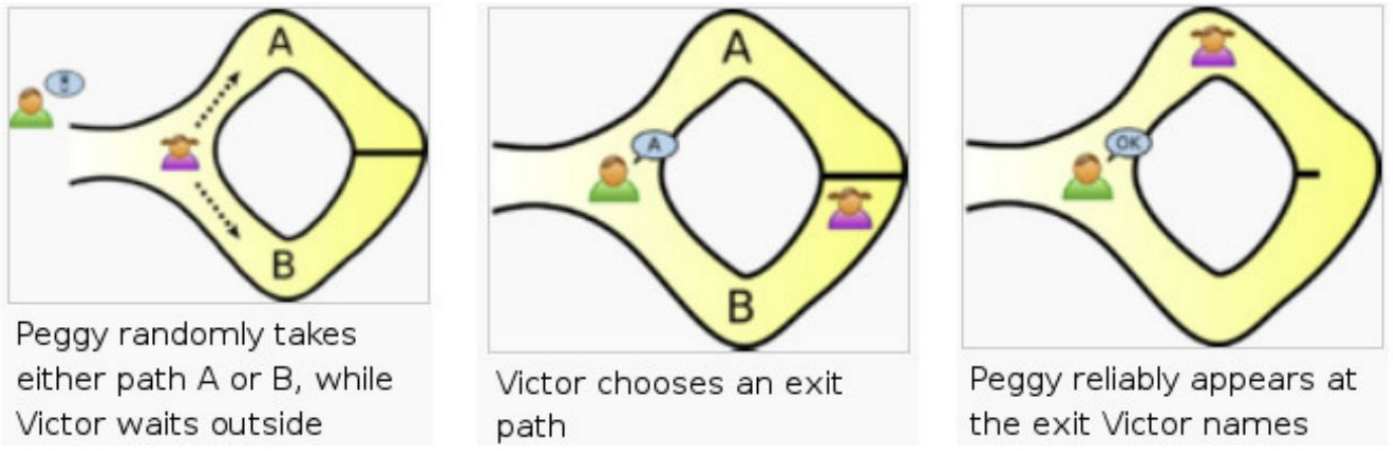
We convince the verifier that we know the password without revealing it.

Nuclear Warhead Verification

We play a game where I do not reveal the number of marbles in my bucket or in my hand but at the end I can show you the total number of marbles. After playing a couple of times, if what I claim is not true (I have a different number of marbles in some other place) the chance of me fooling you is really small. This will never yield 100% certainty but we can play the game until the probability of you fooling me is negligable.



Ali Baba's Cave



1. Peggy enters the cave without Victoring seeing which path she is going.
2. Victor chooses an exit path A or B
3. Peggy leaves through the correct place, Peggy might have to open the door or not open it

After we playing multiple games, the chances that Peggy has luck to never open the door decrease. The chance of her succeeding everything is $(½)^n$ for n rounds of the game.

Zero Knowledge Proofs: The Fiat-Shamir Identification Protocol

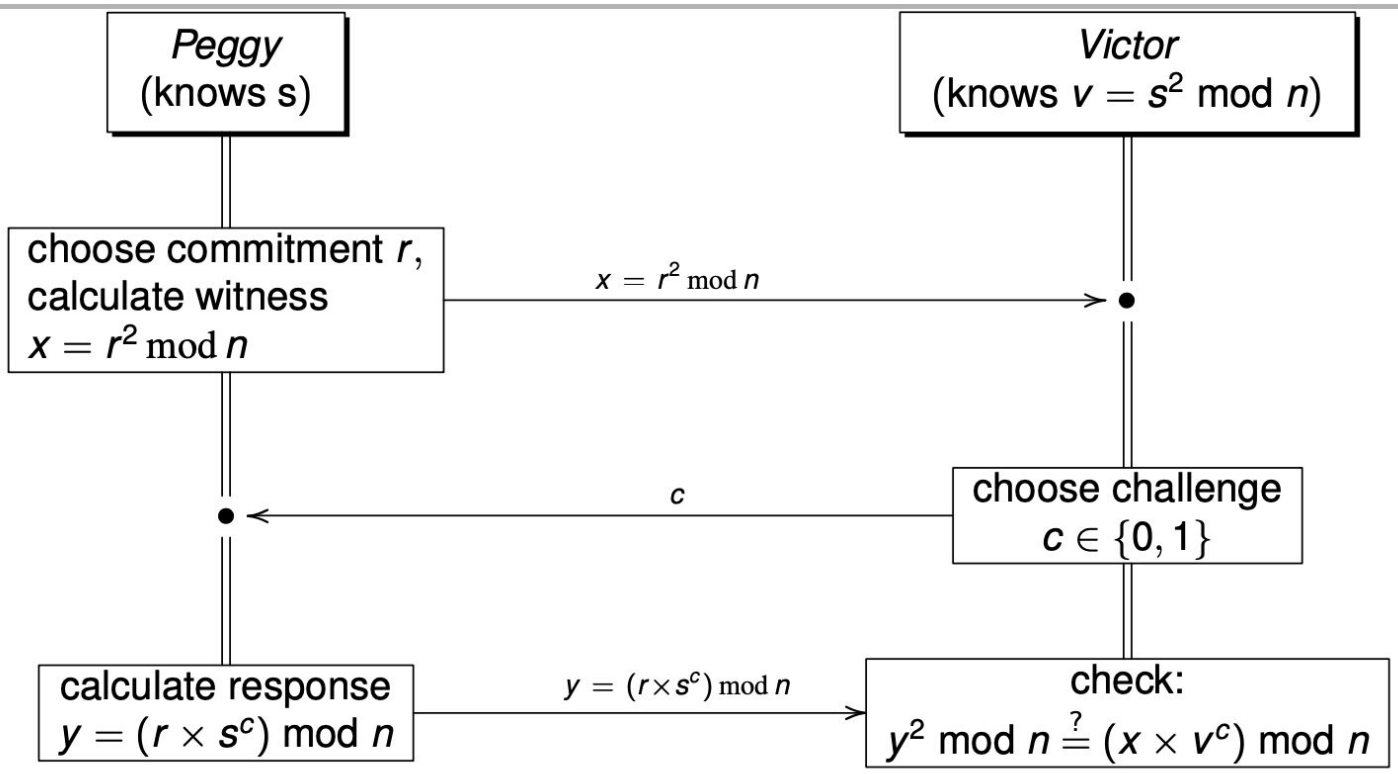
The point of a zero-knowledge protocol is that the challenger learns nothing more than the original claim which is "The prover knows the secret". Evidently, in symmetric cryptography both know the secret but in asymmetric cryptography we must prove that we know the private key we claim we know. A way to do this is via the Fiat-Shamir Identification Protocol.

- Three principals:
  - **Prover Peggy**,
  - **Verifier Victor** and
  - **Trusted Third Party Trent**.
- Setup:
  - Trent chooses two large prime numbers p and q to calculate $n = p \times q$.
  - n is announced to the public, whereas p and q are kept secret.
  - Peggy chooses a **secret number** s between 1 and $n - 1$, and calculates $v = s^2 \bmod n$.
    Peggy keeps s as her **private key** and registers v as her **public key** with the third party.
- Victor knows $v = s^2 \bmod n$, but does not know s.
- Squaring modulo n is easy to compute but square root modulo n is probably not (we believe...).
- **Goal**: Peggy wants to convince Victor that she knows the secret s but Victor should not learn s!

- Verification of Peggy by Victor then proceeds in 4 steps:
  1. Peggy chooses a random number r between 0 and $n - 1$.
     r is called the **commitment**.
     Peggy then calculates the **witness** $x = r^2 \bmod n$ and sends it to Victor.
  2. Victor sends the **challenge** c to Peggy, where c is either 0 or 1.
  3. Peggy calculates the **response** $y = (r \times s^c) \bmod n$ and sends it to Victor to show that she knows her private key s modulo n.
     She claims to be Peggy.
  4. Victor calculates $y^2 \bmod n$ and $(x \times v^c) \bmod n$. If these values are congruent, then Peggy either knows the value of s (she is honest) or she has calculated the value of y in some either ways (dishonest) because in modulo n arithmetic we actually have that

$$y^2 =_n (r \times s^c)^2 =_n r^2 \times s^{2c} =_n r^2 \times (s^2)^c =_n x \times v^c$$

- The 4 steps constitute a **round**.
- The verification is repeated several times with the value of c equal to 0 or 1, chosen randomly.
- Peggy must pass the test in each round to be verified: if she fails one single round, the process is aborted.



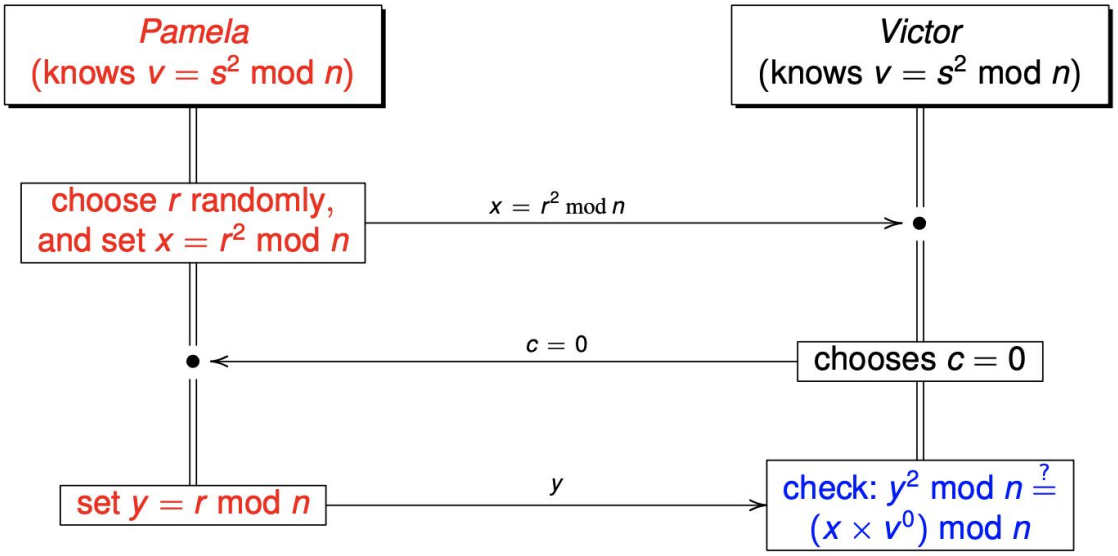
If the check

$$y^2 =_n (r \times s^c)^2 =_n r^2 \times s^{2c} =_n r^2 \times (s^2)^c =_n x \times v^c$$

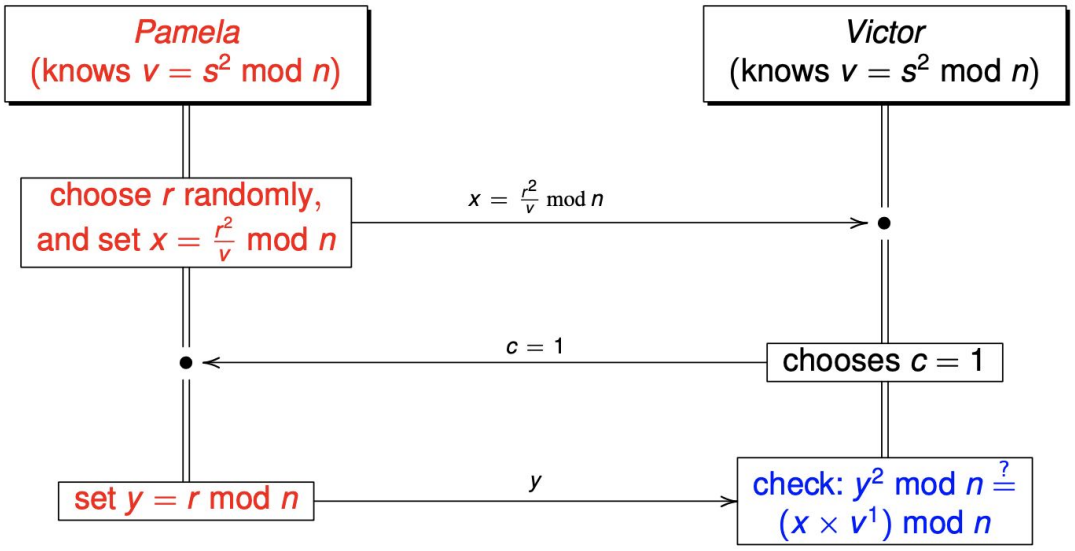
returns a yes, then verification is probable; otherwise, the process is aborted.

As with previous zero-knowledge protocols, we must do it multiple times to ensure that verification is true. All these steps are one round, we must execute multiple rounds.

We can try to cheat by changing our behaviour towards the verifier. In the case that we guess that Victor is going to chose 0, Pamela, out impostor will do:



If Pamela thinks that Victor is going to chose 1:



So, Pamela must find numbers x and y such that $x \times v =_n (x \times s^2)$.
Choose y randomly and then set $\frac{r^2}{v} \bmod n$ (division modulo n is also easy!).

She knows how to set up the numbers when she guesses 0 or 1, but she has to guess.

- Pamela has a strategy to cheat if $c = 0$:

Choose r randomly, set $x = r^2 \bmod n$.

- Pamela has a strategy to cheat if $c = 1$:

Choose r randomly, set $x = \frac{r^2}{v} \bmod n$.

- If $c \in \{0, 1\}$ is randomly chosen, Pamela has thus a chance of $\frac{1}{2}$ to cheat.
- If Victor accepts only after n successful rounds, the chance to cheat is only $\frac{1}{2^n}$.
- We can conclude that

Pamela has no strategy to cheat for unpredictable c.

Although victor may want to learn the secret s, we can conclude that **victor learns nothing except teh proved statement.**

Social Engineering

Social engineering: is the psychological manipulation of people into performing actions or into providing valuable information or access to confidential information.

Or *exploiting the trust people place on electronic systems.*

Why try to break a lock when I can walk behind someone?

Human-based social engineering: we can pretend to be someone else (impersonation) to gain some information. Such calling in the name of a third party, someone important or tech support needing some help.

Computer-based social engineering: we can use features of computers to trick humans into disclosing information like email attachments, websites, pop-ups or fake wireless networks.

- Example: USBs infected with malware which is then put into a computer and effects every computer on the network.