# *CSCI361*
# Computer Security

## Public Key Cryptography I
## Introduction and knapsacks

## Outline

- Drawbacks of symmetric key crypto.
- **P**ublic **K**ey **C**ryptography (PKC).
- Assessing security.
- One-way trapdoor functions.
- Knapsacks.
  - For encryption.
  - Super-increasing knapsacks.
  - Trapdoor knapsacks.
- Finding inverses, GCD’s.
- The Euclidean and Extended Euclidean Algorithms.

## Drawbacks of symmetric key crypto

- So far we have studied **symmetric key cryptosystems** were the transmitter and receiver have the same key.
- They are also called
  - Private key cryptosystems
  - Secret key cryptosystems
  - Conventional cryptosystems

  but we will generally stick to the term symmetric.
- **Key management**, that is, generating good keys, distributing and storing them in a secure way, is a bottleneck in symmetric key cryptography.
- **Example:** For a network of N computer terminals, with pairwise secret keys, the total number of secret keys is N(N-1)/2. For N=100 there are 4950 keys, with 99 at each terminal and two copies of each across the network.

- Another significant drawback of symmetric key cryptography is that the encryption is not identified with an individual, it doesn’t provide **authenticity**. We will later see they are not they cannot directly provide what are referred to as *digital signatures*. *S*ervices such as **non-repudiation** cannot be achieved.
  - A **non-repudiation service** provides Alice protection against Bob later denying that some communication exchange took place. It allows the elements of the communication exchange to be linked with the transmitter.
  - In such a service Alice will have irrefutable evidence to support her claim.
- In symmetric key cryptography the secret information is shared between Alice and Bob, so whatever Alice can do Bob can do too.
  - To provide a non-repudiation service a **trusted authority** is required.

## Public key cryptography

- Diffie-Hellman (1975).

- **Public Key Cryptography (PKC)** provides solutions to both of the problems mentioned.

- In a PKC:
  - The encryption and decryption keys are different.
  - The decryption key cannot be deduced from the encryption key.

- That is, keys come in pairs, **(z, Z)**, where **z** is the private and **Z** is the public component of the key.

- Each pair of keys satisfies the following two properties:
  - A plaintext encrypted with **Z** can be decrypted with **z**. That is **z** determines the inverse of the encryption with key **Z**.
  - Given a public key **Z** it is *computationally infeasible* to discover the corresponding decryption key **z**.

- The first property is needed in all cryptosystems: Decryption is the inverse of encryption.

$$\mathbf{D_z(E_Z(X)) = X}$$

- In secret key cryptography **z** can be easily obtained from **Z**.
- **Example:** In DES decryption sub-keys are the same as encryption sub-keys but applied in reverse order.
- Once **Z** is known, **z** can be calculated and algorithms $\mathbf{E_Z}$ and $\mathbf{D_z}$ are both known. In PKC, knowledge of $\mathbf{E_Z}$ does not imply knowledge of $\mathbf{D_z}$.

- A PK cryptosystem can provide *confidentiality*, because only the receiver can decrypt and find the plaintext, and *authenticity*, because only the sender can create such a cryptogram.

- In a PKC, user Alice has a pair of keys.
  - *Public component* $\mathbf{Z_A}$ generates a public transformation $\mathbf{E_{Z_A}}$ for encryption.

  - *Private component* $\mathbf{z_A}$ generates a private transformation $\mathbf{D_{z_A}}$ for decryption.

- A public directory contains a list of public keys, including a row for Alice.

| | |
|---|---|
| Alice | $Z_A$ |
| Bob | $Z_B$ |
| Fred | $Z_F$ |
| Oscar | $Z_O$ |
| … | |
| … | |

- If **Bob** wants to send a message **X** to **Alice**, he checks the public directory to find Alice's public key, and forms the following cryptogram:

$$Y = E_{Z_A}(X)$$

- **Alice** can recover the message using her private key, as in

$$D_{z_A}(Y) = D_{z_A}(E_{Z_A}(X))$$
$$= X$$

## Assessing security in PK systems

- Since $\mathbf{E_Z}$ is public the enemy can choose a plaintext and find the corresponding ciphertext. Hence **a PKC should be assessed under chosen-plaintext attack.**

- How can we construct a PKC which is resistant to such an attack?
  - PKC's can be realized by using **trapdoor one-way** functions.

## One-way trapdoor functions

- A function *f* is called **one-way** if
  - for all **X** finding ***f*(X)** is easy.
  - knowing ***f*(X)** it is hard to find **X**.
- **Example:** Given n primes $p_1$, $p_2$, … , $p_n$, it is easy to find their product

$$N= p_1p_2\ldots p_n.$$

- Given a large number N it is difficult to find its prime factors.
- A **trapdoor** is a piece of knowledge which makes it easier to find **X** from ***f*(X)**.
- A **trapdoor one-way function** is a function which looks like a one-way function but is equipped with a secret **trapdoor**. If this secret door is known, the inverse can be easily calculated.

- A trapdoor one-way function can be used to realize a PKC:

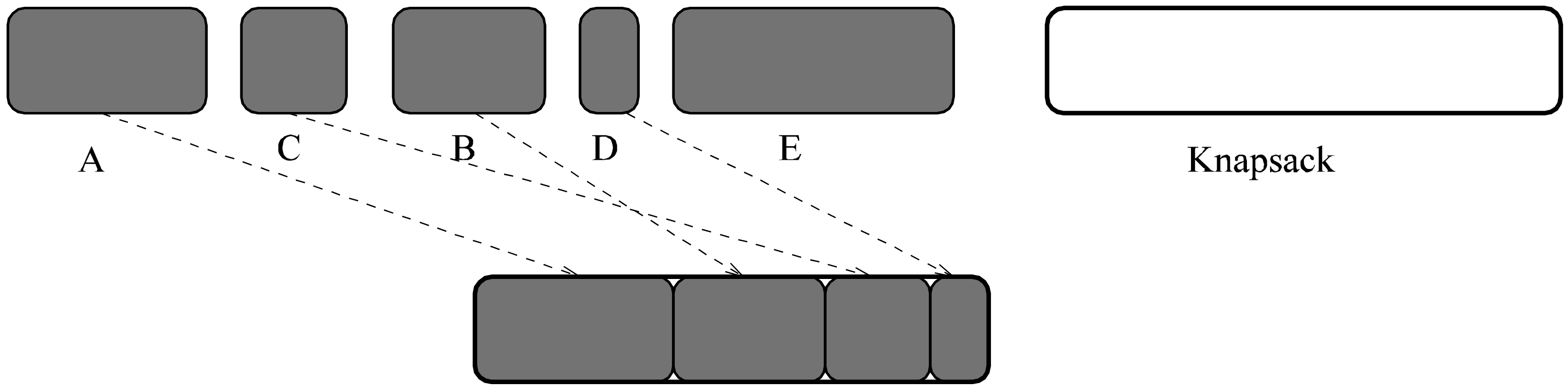
- $\mathbf{E_Z}$ is easy to compute (from the public component of the key **Z**) but is hard to invert.

- Knowledge of **z**, which is the private component of the key (i.e. it is the trapdoor), allows easy computation of $\mathbf{D_z}$ (inverting $\mathbf{E_Z}$).

- We are going to look at two examples of building trapdoors in one-way functions. The first one has failed (knapsacks) and the second one has provided us with the most important PK cryptosystem (RSA).

## Knapsacks

- The **knapsack problem** is derived from the notion of packing an odd assortment of packages into a container:



- *Find a subset of packages that fills the container.*
- In general, there is no efficient way of finding the subset: You have to try all possibilities.

- Alternatively:
  - We are given an ordered set of n positive integers $a_i$, $1 \leq i \leq n$, and a target number T.
  - The ordered set of positive integers is referred to as a *cargo vector*.
  - The knapsack problem is to find a subset $S \subseteq \{1,\ldots n\}$ such that

$$\sum_{i \in S} a_i = T$$

- This is also called the **subset-sum problem**. This is known to be a *difficult* problem as there is no algorithm more efficient than exhaustive search in the general case.

- **Example:** $(a_1,a_2,a_3,a_4)=(2,3,5,7)$ and T=7.
  Solutions: (1,0,1,0), (0,0,0,1).

- Note: In general we may have more than one solution.

# Knapsacks for encryption

- Knapsacks are implemented as **block ciphers**. For a block size of n bits we require a **key** or **cargo vector**, of positive integers $a_i$, $1 \leq i \leq n$, referred to as weights.

  $\mathbf{a} = (a_1, a_2, a_3, ..., a_n)$

- Then to encrypt a message block of n bits:

  $\mathbf{X} = (x_1, x_2, x_3, ..., x_n)$

  we find the number

$$T = \sum_{i=1}^{n} x_i a_i$$

and write it in a binary representation.

- **To decrypt**: The receiver knows T and all $a_i$, so goes through all possible plaintext blocks to find a block that satisfies the condition.
- In this system:
  - **Encryption is easy** and requires n additions.
  - **Decryption is difficult**, even if the key is known.
- Hence the resulting cryptosystem is not useful.
- To construct an acceptable cryptosystem, Merkle & Hellman (1977) used a special case of the knapsack problem, involving *super-increasing knapsacks*, where decryption is easy with the key.

# Super-increasing knapsacks

- A knapsack is super-increasing if each element of the cargo vector is greater then the sum of the preceding elements.
- **Example**: **a**=(1,2,4,8)

  Given T=14 say, it is easy to find X=($x_1$,$x_2$,$x_3$,$x_4$) such that $x_1+2x_2+4x_3+8x_4=14$.

| i | $a_i$ | Total | In? |
|---|---|---|---|
| 4 | 8 | 14 | 1 |
| 3 | 4 | 6 | 1 |
| 2 | 2 | 2 | 1 |
| 1 | 1 | 0 | 0 |

**X=0111**

- At each step the current target value with the largest unchecked element $a_i$ of the cargo vector **a**. If the current target value is larger that element of the cargo vector must be included element (i.e. $x_i=1$) otherwise it cannot be (i.e. $x_i=0$).
- Example: **a**=(171,197,459,1191,2410), T=3798.
- What is $X=(x_1,x_2,x_3,x_4,x_5)$?

| i | $a_i$ | Total | In? |
|---|---|---|---|
| 5 | 2410 | 3798 | 1 |
| 4 | 1191 | 1388 | 1 |
| 3 | 459 | 197 | 0 |
| 2 | 197 | 197 | 1 |
| 1 | 171 | 0 | 0 |

**X=01011**

- As described the super-increasing knapsacks could be used as a symmetric key cryptosystem.
- But super-increasing knapsacks cannot be used directory for public key cryptography, because making the encryption key public *allows everyone to decipher* **X**!
- The idea of Merkle and Hellman (1977) was to start with an easy knapsack and then *disguise* it to *look difficult* for people without knowledge of the trapdoor.
- The trapdoor will only be known by the person who wants to decrypt the cryptogram. It will be their private key.

## Trapdoor knapsacks

- Alices chooses a super-increasing knapsack, $a=(a_1,a_2,\ldots,a_n)$, as her **private key**.
- Then she chooses an integer $m > \sum_{i=1}^{n} a_i$, called the **modulus**, and a random integer **w**, called the **multiplier**, which is relatively prime to **m**. Relatively prime means the largest common factor of **w** and **m** is 1. This condition implies there exists an inverse of **w modulo m**.
  - – Note that if **m** is prime, then **w** can be any number.
- Alice’s public key is **b** where

  **b**$=(b_1,b_2,\ldots,b_n)$ and $b_i=w*a_i \bmod m$

- Alice publishes a permuted version of **b** as her public key. Her secret key is **(a,m,w)**.
- **To encrypt**: Bob sends a message **X** to Alice by computing

$$T = \sum_{i=1}^{n} x_i b_i$$

- **To decrypt**: Alice receives T and …
  - Uses the inverse of **w**, $\mathbf{w^{-1}}$, to calculate

    $R=w^{-1}T \bmod m$.
  - Uses the easy knapsack **a** to find **X**, since **R=a.X**

$$
\begin{aligned}
R &= w^{-1}T \pmod{m} \\
&= w^{-1}\sum_{i=1}^{n} b_i x_i \pmod{m} \\
&= w^{-1}\sum_{i=1}^{n} (w a_i) x_i \pmod{m} \\
&= \sum_{i=1}^{n} (w^{-1} w a_i) x_i \pmod{m} \\
&= \sum_{i=1}^{n} a_i x_i \pmod{m} \\
&= a.X
\end{aligned}
$$

## Brute force attack

- For those who do not know the secret trapdoor, decryption requires an exhaustive search through all $2^n$ possible **X**.

- **Example**:
- Earlier we had: **a**=(171,197,459,1191,2410)
- Let w = 2550 and m = 8443.

  **b**=(5457, 4213, 5316, 6013, 7439) is the public cargo vector.

## Multiple layers and the fall of knapsacks

- The disguising process can be repeated a number of times on the cargo vector to create more and more difficult knapsack problems **($w_1$, $m_1$)**, **($w_2$, $m_2$)**,... . The result is, in general, not equivalent to a single **(w, m)** transformation.
- Adleman (1982) broke the knapsack cryptosystem by taking a public cargo vector and finding a pair **(w', m')** that would convert it back to a super-increasing cargo vector sufficient for decrypting encrypted messages. It didn't haven't to be the same one used as the private key.
- Merkle was confident enough that the multiple layers were still secure to offer a reward of $1000 to anyone who could break a multiple layer knapsack.
- In 1984, Brickell announced the destruction of a knapsack system, with 40 iterations and a hundred weights (elements of the cargo vector), in about one hour of Cray-1 time.
  - Merkle gave him the money ☺

## Finding inverses: The Extended Euclidean Algorithm

- Constructing trapdoors in knapsacks requires finding inverses of **w modulo m**.
- Inverses exists if **w** and **m** do not have any common factor.
- To find $\mathbf{w^{-1}}$ such that $\mathbf{w^{-1}w = 1 \bmod m}$ we will use the Extended Euclidean Algorithm.
  - Before doing so it is instructive to look at the Euclidean algorithm.

## GCD's and the Euclidean Algorithm

- The *greatest common divisor* (GCD) of two integers $n_1$ and $n_2$, not both zero, is the largest integer that divides $n_1$ and $n_2$.
- It is denoted **gcd($n_1$,$n_2$)**.
- **Example**: gcd(30, 15) = 15

  gcd(30, -12) = 6
- We can calculate the gcd using an algorithm of Euclid.

## Euclidean Algorithm

1) Divide the larger number by the smaller and retain the remainder.
2) Divide the smaller original number by the remainder, again retaining the remainder.
3) Continue dividing the prior remainder by the current remainder until the remainder is zero, at which point the last (non-zero) remainder is the greatest common divisor.

- **Example**: gcd(84,49).

  84/49 ➔ remainder 35.

  49/35 ➔ remainder 14.

  35/14 ➔ remainder 7.

  14/7 ➔ remainder 0.

Therefore gcd(84,49)=7.

## Extended GCD for integers

The Extended GCD Theorem for Integers states:

Given integers $n_1$ and $n_2$, not both zero, there exist integers a and b such that

$$gcd(n_1,n_2)=a*n_1+b*n_2$$

- These integers are not necessarily unique though.
- **Example**:

  gcd(15,12) = 3 = (+1)*15+(-1)*12
  = (+1-12)*15+ (-1+15)*12
  = (-11)*15+(+14)*12

If gcd(w,m)=1 then it means that we can find the inverses $\mathbf{n_1}$ **mod** $\mathbf{n_2}$ and $\mathbf{n_2}$ **mod** $\mathbf{n_1}$.

$$gcd(n_1,n_2)=a*n_1+b*n_2=1$$

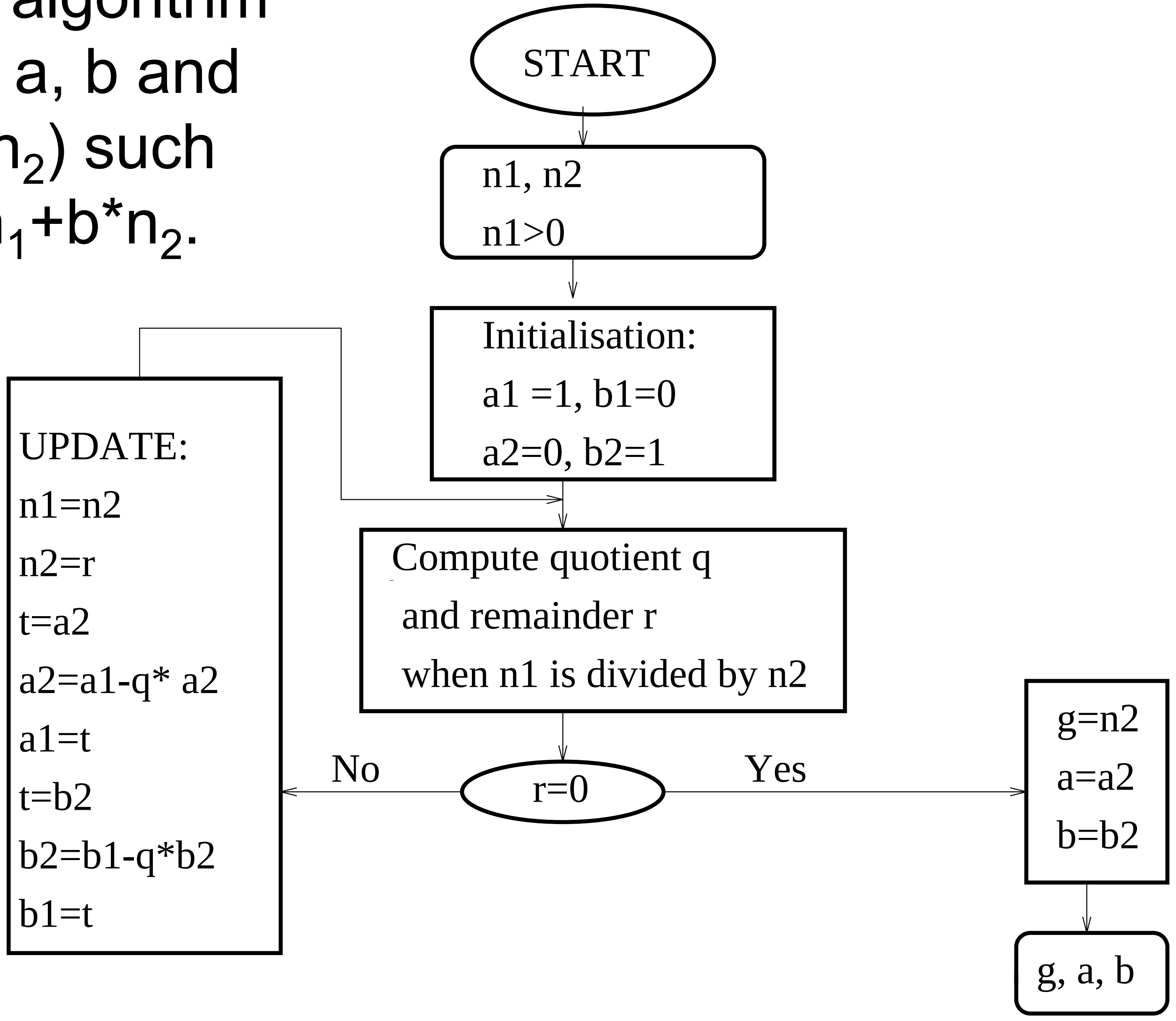
- **Example**:

  gcd(65,14) = 1 = (-3)*65+(14)*14

  ➔ 14*14=1 (mod 65)

- The Extended Euclidean algorithm calculates a, b and $g=\gcd(n_1,n_2)$ such that $g=a*n_1+b*n_2$.

The extension →



## Find gcd(39,11) and a,b : 39a+11b=gcd(39,11)

Initialise

| $n_1$ | $n_2$ | $r$ | $q$ | $a_1$ | $b_1$ | $a_2$ | $b_2$ |
|---|---|---|---|---|---|---|---|
| 39 | 11 | 6 | 3 | 1 | 0 | 0 | 1 |
| 11 | 6 | 5 | 1 | 0 | 1 | 1 | -3 |
| 6 | 5 | 1 | 1 | 1 | -3 | -1 | 4 |
| 5 | **1** | 0 | 5 | -1 | 4 | **2** | **-7** |

**gcd(39,11)=1** **1=39*2+11*(-7)**

## Example: PKC using a trapdoor knapsack

- Secret key **a**=(2, 5, 10, 21).
- Trapdoor: **m**=39>38 (sum of weights).
  - Random w, **w**=15.
  - gcd algorithm, gcd(39,15)=3≠1
  - Another w, **w**=11.
  - gcd(39,11)=1
  - Inverse of 11 mod 39 = 32

    (32 = -7+39)

- Using **w**=11 and **m**=39 disguise **a**.

  $b_1$=2*11=22 (mod 39)

  $b_2$=5*11=16 (mod 39)

  $b_3$=10*11=32 (mod 39)

  $b_4$=21*11=36 (mod 39)
- Public key: **b**=(22, 16, 32, 36).

- Secret key: **a**=(2, 5, 10, 21).

  **(w,m)**=(11, 39)

(remember also that $\mathbf{w}^{-1}$=32)

- To decrypt a message T=48…
  - Find R = 48*32 mod 39 = 15
  - Use the cargo vector **a** to decrypt R and find X=(0, 1, 1, 0).
  - We see this agrees with the public key version too (16+32=48).
  - In useful sized examples it would be not trivial to find the solution using just the public key, although it is easy to check the answer using it.

## PKC progress…

- Since 1976, many PKC have been proposed. Many of these are broken and proven to be insecure.
- Among the ones that are considered secure, many are impractical because of either large key or large message expansion.
- PKC algorithms are, however, all slow and unsuitable when fast encryption is needed. However, they can provide high security and can be used when low processing rates are acceptable or when high security is needed.
- Remember that a PKC can be used for:
  - confidentiality (secrecy).
  - authentication (digital signatures).
- Two algorithms that can easily be used for both *secrecy and authentication* are RSA and ElGamal.
  - We are going to look at RSA next.

*CSCI361*

Computer Security

# Public Key Cryptography II
RSA

## Outline

- RSA.
  - Encryption and decryption.
- Using RSA.
- Choosing p and q.
- Implementation considerations.
- Some comments on factoring.
- Finding and testing primes.
- Fast exponentiation.
- Assessing the security of RSA.

## RSA

- The RSA Public--Key Cryptosystem (**R**ivest, **S**hamir and **A**dleman (1978)) is the most popular and versatile PKC.

- It is the *de facto* standard for PKC.
- It supports *secrecy and authentication* and can be used to produce *digital signatures*.
- RSA uses the knowledge that it is *easy* to find primes and multiply them together to construct composite numbers, but it is *difficult* to factor a composite number.

# The Euler phi Function

**For $n \geq 1$, $\phi(n)$ denotes the number of integers in the interval [1, n] which are relatively prime to n. The function $\phi$ is called the Euler phi function (or the Euler totient function).**

Fact 1. The Euler phi function is multiplicative, i.e. if $\gcd(m, n) = 1$, then $\phi(mn) = \phi(m) \times \phi(n)$.

Fact 2. For a prime p and an integer $e \geq 1$, $\phi(p^e) = p^{e-1}(p-1)$.

- From these two facts, we can find $\phi$ for any composite n if the prime factorization of n is known.

## The Euler phi Function

$$\phi(n) = \left|\{x : 1 \le x \le n \quad and \quad \gcd(x,n) = 1\}\right|$$

- $\phi(2) = |\{1\}| = 1$
- $\phi(3) =|\{1,2\}| = 2$
- $\phi(4) = |\{1,3\}| = 2$
- $\phi(5) = |\{1,2,3,4\}| = 4$
- $\phi(6) = |\{1,5\}| = 2$

- $\phi(37) = 36$
- $\phi(21) = (3–1)\times(7–1) = 2\times6 = 12$

## The algorithm

1. Choose two primes p and q. Compute $n = pq$ and $m=\phi(n)= (p-1)(q-1)$.
   - $\phi(n)$ is Euler's totient function: It is the number of positive integers less than n that are relatively prime to n.
2. Choose e, $1 \leq e \leq m - 1$, such that $\gcd(e,m)=1$.
3. Finds d such that $ed=1 \bmod m$.
   - This is possible because of the choice of e.
   - d is the multiplicative inverse of e modulo m and can be found using the extended Euclidean (gcd) algorithm.
4. The **Public key** is **(e, n)**.

   The **Private key** is **(d, p, q)**.

# Encryption and decryption

- Let **X** denote a plaintext block, and **Y** denote the corresponding ciphertext block.
- Let **($z_A$, $Z_A$)** denote the private and public components of Alice's key.

- If Bob want to encrypt a message **X** for Alice. He uses Alice's public key and forms the cryptogram:

$$\mathbf{Y = E_{Z_A}(X) = X^e \bmod n}$$

- When Alice wants to decrypt **Y**, she uses the private component of her key **$z_A$=(d,n)** and calculates

$$\mathbf{X = D_{z_A}(Y) = Y^d \bmod n}$$

- **X** and **Y** are both integers in {0, 1, 2, …, n-1}.

- **Example**: Choose p=11 and q=13.

  n=11*13=143

  m=(p-1)(q-1)=10*12=120

  e=37 ➔ gcd(37,120)=1

Using the gcd algorithm we find d such that ed=1 mod 120; d=13 → de=481.

$X,Y \in \{0,1,\ldots,142\}$

- **To encrypt**: Break the input binary string into blocks of *u* bits, where $2^u \leq 142$, so we choose u=7.

  In general there is some agreed **padding scheme** from the message space into the space on which a "single run" of the cipher can act.

  Optimal Asymmetric Encryption Padding (OAEP) is often used.
- For each 7-bit block **X**, a number between 0 and 127 inclusive, we calculate the ciphertext as $\mathbf{Y=X^e}$.
- For X=2=(0000010) we have

  $E_Z(X)=X^{37}=12 \pmod{143} \rightarrow Y=0001100$
- **To decrypt**: $X=D_Z(Y)=12^{13}=2 \pmod{143}$

## An RSA public directory

| User | (n,e) |
|---|---|
| Alice | (85,23) |
| Bob | (117,5) |
| Fred | (4757,11) |

- An important property of the RSA algorithm is that encryption and decryption are the same function: both exponentiation modulo n.

$$E_{Z_A}(D_{z_A}(X)) = X$$

- Example:
  - First decrypt: $2^{13} = 41 \bmod 143$
  - Then encrypt: $41^{37} = 2 \bmod 143$

- This is the basis of using RSA for authentication.

# Using RSA

- The RSA system can be used for:

1. **Confidentiality**: To hide the content of a message **X**, A sends $\mathbf{E_{z_B}(X)}$ to B.
2. **Authentication**: To ensure integrity of a message **X**,
   - Alice *signs* the message by using her decryption key to form $D_{z_A}(X)$ and sends $(X, D_{z_A}(X)) = (X, S)$ to Bob.
   - When Bob wants to *verify the authenticity* of the message:
     - He computes $X' = E_{z_A}(S)$.
     - If X'=X the message is accepted as authentic and from Alice.
   - Both **message integrity** and **sender authenticity** are verified.
     - This is true because even one bit change to the message can be detected, and because $\mathbf{z_A}$ is known only to Alice.
   - This method is inefficient. We will see later that Alice may compute a hash value of **X** and then apply $D_{z_A}$ to the result.
   - We will talk about Digital Signatures later anyway.

## 3. **Secrecy and authentication**:

- If we need secrecy and authentication the following can be used:
- A sends $Y=E_{Z_B}(D_{z_A}(X))$ to B.
- B recovers

$$X=E_{Z_A}(D_{z_B}(Y))$$

This scheme provides *non-repudiation* if Bob holds on to $D_{z_A}(X)$. That is, Bob is protected against Alice trying to deny sending the message.

## Choosing p and q

- The main conditions on p and q are:
  - They must be at least 100 decimal digits long (about 330 bits).
  - They must be of similar say, say both 100 digits.
- To choose each number the user does the following:
  - Randomly choose a random number b which is 100 digits long, or whatever length is appropriate.
  - Checks to see if b is prime. Usually using a probabilistic primality testing algorithm.
  - If b is not prime, choose another value for b.

# RSA implementation considerations

- Relative to DES or AES;
  - RSA is a much slower algorithm.
  - RSA has a much larger secret key.
- Storage of p and q requires about 330 bits each, n is about 660 bits.
- Exponentiation is relatively slow for large n, especially in software.
- It can be shown that multiplication needs m + 7 clock pulses if n is m-bits in length.

- Approximate relative speeds (this is a few years old!)
  - Hardware: n is 507 bits: 220 kbits/sec
  - Software: n is 512 bits:  11 kbits/sec

## Some comments on factoring

- The most powerful known (pretty much general purpose) factoring algorithm is the *general number field sieve*.
- It uses

$$O\left(\exp\left(\frac{64}{9}\log n\right)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}}\right)$$

steps to factor a number n.

- One of the best factoring results is for a 663-bit number (RSA-200). This was factored using a general number field sieve (announced May 9 2005). This took several months of work undertaken by a system of 80 AMD Opteron processors (~2.6GhZ).

- Today for sensitive applications, a 1024 bit modulus, and in some cases a 2048 bit modulus, are considered necessary.

## Finding primes

- An algorithm to generate all primes does not exist.
- However, given a number n, there exist efficient algorithms to *check whether it is prime or not*. Such algorithms are called **primality testing algorithms**.
- As mentioned earlier, prime generation for RSA is basically just a matter of guessing and testing.
- Primality Testing: Deterministic algorithms for proving primality are non-trivial and are only advisable on high performance computers. Probabilistic tests allow an *educated guess* as to whether a candidate number is prime or not.
  - This means that the probability of the guess being wrong can be made arbitrarily small.

## Lehman's test

- Let **n** be an odd number. For any number **a** define

$$e(a,n) = a^{\frac{n-1}{2}} \bmod n$$
$$G = \{e(a,n) : a \in Z_n^*\}$$

  where $Z_n^*=\{1,2,\ldots,n-1\}$.
- **Example**:  n=7

  $2^3=1$, $3^3=6$, $4^3=1$, $5^3=6$, $6^3=6$ $\rightarrow$ G={1,6}

- **Lehman's theorem**:

  If n is odd, G={1,n-1} if and only if n is prime.

**Example**: n=15 isn't prime: (n-1)/2=7

$2^7=8 \bmod 15$, $3^7=12 \bmod 15$

**Example**: n=13 (prime): (n-1)/2=6

$2^6=-1 \bmod 13$, $3^6=1 \bmod 13$, $4^6=1 \bmod 13$,

$5^6=-1 \bmod 13$, $6^6=-1 \bmod 13$, $7^6=-1 \bmod 13$,

$8^6=-1 \bmod 13$, $9^6=1 \bmod 13$, $10^6=1 \bmod 13$,

$11^6=1 \bmod 13$, $12^6=1 \bmod 13$.

- Thus, we have the following test:

```
if (gcd(a,n) >1) return('composite')
else
    if ($a^{(n-1)/2}=1$) or ($a^{(n-1)/2}=-1$)
        return('prime_witness')
    else
        return('composite')
```

If for a given **n** the test returns prime_witness for 100 randomly chosen **a**, then the probability of **n** not being not prime (i.e. being a composite disguised as a prime) is less than $2^{-100}$.

# Fast exponentiation

- Exponentiation can be performed by repeated multiplication. In general, we can use the *square and multiply* technique.
- To calculate $X^{\alpha}$:

1. Write $\alpha$ in base two:

   $\alpha= \alpha_0 2^0+ \alpha_1 2^1+ \alpha_2 2^2+\dots \alpha_{n-1} 2^{n-1};$
2. Calculate $X^{2^i}$, $1\leq i\leq n-1$.
3. Use $X^{\alpha}=(X^{2^0})^{\alpha_0}*(X^{2^1})^{\alpha_1}\dots(X^{2^{n-1}})^{\alpha_{n-1}}$ and

   Multiply the $X^{2^i}$ for which $\alpha_i$ is not zero.

- **A partial example**: n=179, e=73.
  $X=2 \rightarrow Y=2^{73} \bmod 179$.
  $73=64+8+1=2^6+2^3+2^0$

  $Y=2^{64+8+1}=2^{64}*2^8*2^1$

This is only a partial example because we haven't looked at calculated the elements of the last line.

**Precomputation**:

$X^2=X*X$

$X^4= X^{2^2}=X^{2*}X^2$

…

$X^{2^{n-1}}= X^{2^{n-2}}* X^{2^{n-2}}$

**This is a total of n-1 multiplications, all mod N**

- **Example**: N=1823, n=$\log_2 1822$=11.
  - Calculate Y=$5^{375}$ mod N
- Precomputation:

| | | | | | |
|---|---|---|---|---|---|
| $X^1$ | 5 | $X^2$ | 25 | $X^4$ | 625 |
| $X^8$ | 503 | $X^{16}$ | 1435 | $X^{32}$ | 1058 |
| $X^{64}$ | 42 | $X^{128}$ | 1764 | $X^{256}$ | 1658 |
| $X^{512}$ | 1703 | $X^{1024}$ | 1639 | | |

- Never store a number larger than N!
- Never multiply two numbers large than N!

  375=256+64+32+16+4+2+1.

$5^{375}$=5*25*625*1435*1058*42*1658

= 591 mod 1823

There are various other tricks for calculating powers too, but we aren’t going to look at them here!

# A weakness in RSA

- In RSA not all the messages are concealed, i.e. the plaintext and ciphertext are the same.
- **Example**: n=35=5*7, m=4*6.
- X=8.
- Y=$8^5$ mod 35=8

- For any key, at least 9 messages will not be concealed. But for n $\geq$ 200 or so, this is not important.

- However if *e* is poorly chosen, less than 50% of the messages are concealed.
- **Example**: n=35, e=17.

  {1,6,7,8,13,14,15,20,21,22,27,28,29,34} are not concealed. ☹

  It is less likely that a system will have this problem if the primes are **safe**.

  A prime is **safe** if p=2q+1, where q is a prime itself.

## Assessing the security of RSA

- The security of the private component of a key depends on the difficulty of factoring large integers.
- **Justification**:
  - Let Z=(e,n). If n can be factorised then an attacker can find
    - n=pq
    - m=(p-1)(q-1)
  - … and use the gcd algorithm to find the private key, $d=e^{-1} \bmod m$.
- No efficient algorithm for factoring *is known*.

- So knowing n = pq does not yield p or q.
- This implies that m=(p-1)(q-1) cannot be found and d cannot be computed.

- **Finding secret exponent**: If an attacker knows X and Y= $D_z(X)$ to find d they must solve

  $$X = Y^d \bmod n \quad \text{or} \quad d = \log_Y X$$

- **Finding plaintext**: If the attacker knows Y and e to determine X they must take roots modulo a composite number: i.e. they need to solve $Y = X^e$.

- It is important to note that the security of RSA **is not provably** equivalent to the difficulty of factoring. That is, it might be possible to break RSA without factoring n.

## Important attacks

- It is sufficient for the cryptanalyst to compute $\phi(n)$.
- Knowledge of n and $\phi(n)$ gives two equations:

$$n = pq$$

$$\phi(n) = (p-1)(q-1)$$

This system can be solved, for p say, by solving a quadratic equation:

$$p^2-(n-\phi(n)+1)p+n=0$$

- **Some equivalent results**:
  - Any algorithm that can compute the decryption exponent can be used as a subroutine in a probabilistic algorithm that factors n.
  - Any algorithm that computes *parity(Y)* can be used as a subroutine in an algorithm that computes the plaintext X.

# Weak implementations

- Some implementations allow attacks:

1. **Common modulus attack**: Consider a group of users who's public keys consists of the same modulus and different exponents.
   - If an intruder intercept two cryptograms where
     - They are encryptions of the same message with different keys.
     - The two encryption exponents do not have any common factor. Then the attacker can find the plaintext.

- Let us consider why:

- The enemy knows $e_1$, $e_2$, N, $Y_1$ and $Y_2$, and furthermore that $Y_1 = X^{e_1} \bmod N$ and $Y_2 = X^{e_2} \bmod N$

- Since $e_1$ and $e_2$ are relatively prime, the Extended Euclidean algorithm can be used to find **a** and **b** such that $ae_1 + be_2 = 1$.

But then $Y_1^a Y_2^b = X^{ae_1} * X^{be_2} = X^{ae_1+be_2} = X$

- **Using a common modulus among a number of participants is not advisable!**

2. **Low exponent attack**: If the encryption exponent is small, encryption and signature verification will be faster.
   - However, if the enemy can find **e(e+1)/2** linearly independent messages encrypted (signed) with the same exponent, then they can also find the message. The attack does not work if the messages are unrelated.
   - Pad messages with random strings to ensure that they are not dependent.
3. **Low decryption exponent**: If the decryption exponent is less that N/4 and e<N then d can be found.

## RSA in real-life

1. If there is a small range of possible messages,
   - For example, messages are numbers between 0 and 1,000,000.
   - Then given a ciphertext, the plaintext can be found by exhaustively searching all plaintext.
2. The *Multiplicative property* of RSA can be used to find the plaintext.

$$C_1 = m_1^e \bmod N$$
$$C_2 = m_2^e \bmod N$$
$$C_1C_2 = m_1^e\, m_2^e = (m_1m_2)^e \bmod N$$

## An attack against RSA

- Suppose *m* is *l* bits, $m \le 2^l$.
- Suppose $m=m_1m_2$ $m_1, m_2 \le 2^{l/2}$.
- The attacker knows $c=m_1^e m_2^e \bmod N$
- … and builds a sorted database:

  $\{1^e, 2^e, 3^e, \dots, (2^{l/2})^e\} \bmod N$
- For a given *c*, the attacker searches the database for $c/i^e=j^e \bmod N$.
- This will take at most $2^{l/2}$ steps.
- The cost if $2^{l/2}\log N$ space is affordable is…

$$O\left(2^{l/2+1} * \left(\frac{l}{2} + \log^3 N\right)\right)$$