Public Key Cryptography

All **secret key** algorithms & **hash** algorithms do the same thing

but **public key** algorithms look **very different** from each other.

What is common among all of them is that

each participant has ***two keys***, **public** and **private**,

and most of them are based on ***modular arithmetic***.

Modular Arithmetic

x mod n is the ***remainder*** of ***x*** when divided by ***n***.

e.g., 8 mod 10 = 8, 18 mod 10 = 8, 24 mod 10 = 4
8 mod 7 = 1, 18 mod 7 = 4, 24 mod 7 = 3

- ***Addition:***

Example: addition mod 10

8 + 8 = 6,
1 + 9 = 0,
7 + 6 = 3

See Fig. 6-1 for addition mod 10 Table:

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	0
2	2	3	4	5	6	7	8	9	0	1
3	3	4	5	6	7	8	9	0	1	2
4	4	5	6	7	8	9	0	1	2	3
5	5	6	7	8	9	0	1	2	3	4
6	6	7	8	9	0	1	2	3	4	5
7	7	8	9	0	1	2	3	4	5	6
8	8	9	0	1	2	3	4	5	6	7
9	9	0	1	2	3	4	5	6	7	8

Figure 6-1. Addition Modulo 10

Encryption: Addition mod 10 can be used for encryption of digits.

Add **k**, a **secret key** between 1-9, to each digit.

Example: if **k** = 7, then 1987 is ***encrypted*** to 8654.

Decryption: Add **-k,** the ***additive inverse*** of **k**, to each digit.

An additive inverse of **x** is the number you'd have to add to **x** to get **0**.

Example: if **k** = 7, then **-k** is 3 since 7+3 = 0

Thus 8654 will is decrypted to 1987.

In the above table (Fig. 6-1), each "0" is the intersection of

k and **-k**, e.g., 0 is the intersection of 3 and 7.

- ***Multiplication:***

Example: multiplication mod 10 :

8 x 8 = 4, 1 x 9 = 9 , 7 x 6 = 2

See Fig. 6-2 for multiplication mod 10 Table:

·	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	0	2	4	6	8
3	0	3	6	9	2	5	8	1	4	7
4	0	4	8	2	6	0	4	8	2	6
5	0	5	0	5	0	5	0	5	0	5
6	0	6	2	8	4	0	6	2	8	4
7	0	7	4	1	8	5	2	9	6	3
8	0	8	6	4	2	0	8	6	4	2
9	0	9	8	7	6	5	4	3	2	1

Figure 6-2. Multiplication Modulo 10

Encryption: multiplication by 1, 3, 7, 9 works as a cipher since it performs 1-1 mapping.

Example: if **k** = 7, then 1987 is encrypted to 7369

Decryption: is done by multiplying each digit by $\mathbf{k}^{-1}$ **,**

the *multiplicative inverse* of **k**.

It is the number to multiply by **k** to get **1**.

Example: if **k** = 7, then $\mathbf{k}^{-1}$ is 3 since 7x3 = 1

In the above table (Fig. 6-2),

each "**1**" is the intersection of **k** and $\mathbf{k^{-1}}$.

Note that only **{1,3,7,9}** have multiplicative inverse mod 10.

- What is so special about the set **{1,3,7,9}**?

  These numbers are ***relatively prime*** to 10,

  i.e., they do not share with 10 any common factors other than 1.

  Note that 9 is not a prime number but it is relatively prime to 10.

- How many numbers less than ***n*** are relatively prime to ***n***?

  This quantity is referred to as:

  Ø(n) and is called the ***totient function***.

  - *If **n** is prime*:

    then {***1,2, ..., n-1***} are all relatively prime and thus ***Ø(n) = n-1***.

  - If ***n = p.q*** where ***p*** and ***q*** are two distinct primes,

    then ***Ø(n) = (p-1)(q-1)***.

  Example: for n = 10 = 2.5, Ø(10) =(2-1).(5-1)=1.4=4, which is the set {1,3,7,9}.

- ***Exponentation:***

  Example: exponentiation mod 10

$4^2 = 6$, $8^8 = 6$, $1^9 = 1$, $7^6 = 9$

See Fig. 6-3 for exponentiation mod 10 Table:

x^y	0	1	2	3	4	5	6	7	8	9	10	11	12
0		0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	2	4	8	6	2	4	8	6	2	4	8	6
3	1	3	9	7	1	3	9	7	1	3	9	7	1
4	1	4	6	4	6	4	6	4	6	4	6	4	6
5	1	5	5	5	5	5	5	5	5	5	5	5	5
6	1	6	6	6	6	6	6	6	6	6	6	6	6
7	1	7	9	3	1	7	9	3	1	7	9	3	1
8	1	8	4	2	6	8	4	2	6	8	4	2	6
9	1	9	1	9	1	9	1	9	1	9	1	9	1

Figure 6-3. Exponentiation Modulo 10

Amazing fact about **Ø(n)**:

$$x^{m} \bmod n = x^{m \bmod Ø(n)} \bmod n$$

Since Ø(10)=4, in Fig. 6-3:

$$x^{m} \bmod 10 = x^{m \bmod 4} \bmod 10$$

the ***i^{th} column* is identical to** the ***$i+4^{th}$ column***,

e.g., $1^{st} = 5^{th} = 9^{th}$ and $3^{rd} = 7^{th} = 11^{th}$.

Special case:

if ***$m = 1 \bmod Ø(n)$***, then for any number x,

$x^{m} \bmod n = x \bmod n$.

Example: **For n =10, Ø(10)=4**

Since **m = 9** is = 1 mod 4:

3^9 mod 10 = 3 mod 10 = 3 &

6^9 mod 10 = 6 mod 10 = 6

in general:

for any **x :** **x^9 mod 10 = x mod 10 = x**

An ***exponentiative inverse*** of ***e*** is the number ***d*** such that:

$$e.d = 1 \bmod \text{Ø}(n)$$

Example: For n= 10, Ø(10)=4:

e=3 and d=7 are ***exponentiative inverses*** since:

3.7 = 21 = 1 mod 4

Encrypt/Decrypt:

- To encrypt ***m***: compute $c = m^e \bmod n$
- To decrypt ***c***: compute $m = c^d \bmod n$

Example:

encrypt m = 8: $c = 8^3 = 2$
decrypt c = 2: $m = 2^7 = 8$

Sign/Verify:

- To sign ***m***: compute $s = m^d \bmod n$
- To verify ***s***: compute $m = s^e \bmod n$

Example:

sign $m = 8$: $s = 8^7 = 2$

verify $s = 2$: $m = 2^3 = 8$

In public cryptography:

<*e*, n> is *public key*
&
<d, n> is *private key*

RSA: Rivest, Shamir & Adleman

Key length:

Variable (**long** for ***security***, **short** for ***efficiency***).
Most common values is 512 & 1024 bits.

Block size:

plain text length is variable but **less than** ***key length*** &
cipher text length **equals** ***key length***.

Thus RSA is used for ***encrypting*** **small** amount of data,

e.g., a secret key and then use the secret key cryptography for

encrypting/decrypting **large** amount of data.

RSA Algorithm:

Generate public & private keys pair:

1. Choose two large primes ***p*** and ***q***.
   (Typically 256 bits each & keep them secret).

2. Compute $n = p.q$ & $\text{Ø}(n) = (p-1)(q-1)$.
   (It is very hard to factor ***n*** into ***p*** & ***q***).

3. Choose a number ***e*** that is relatively prime to ***Ø(n).***

4. Find a number ***d*** that is the exponentiative inverse of ***e***
   i.e., e.d = 1 mod Ø(n).

5. The ***public key***: <e,n> & the ***private key***: <d,n>.

Encrypt/Decrypt:

To encrypt a message **m** (**less than** n):

$$c = m^e \bmod n$$

To decrypt c:

$$m = c^d \bmod n$$

This works since:

$$\begin{aligned} c^d \bmod n &= (m^e)^d \bmod n \\ &= m^{e.d} \bmod n \\ &= m \bmod n \quad \text{// since } e.d = 1 \bmod \text{Ø}(n) \\ &= m \quad \text{// since } m < n \end{aligned}$$

Sign/Verify:

To sign a message m (**less than** n):

$$\mathbf{s} = m^d \bmod n$$

To verify s:

$$m = \mathbf{s}^e \bmod n$$

This also works since:

$$\mathbf{s}^e \bmod n = m^{e.d} \bmod n = m \bmod n = m$$

Why is RSA Secure:

Every one knows the public key: <e, n>.

To find the private key <d,n>
you need to know Ø(n) since e.d = 1 mod Ø(n).

To know Ø(n) you need to know p and q since
Ø(n) = (p-1).(q-1).

Thus to break RSA you should know:

how to factor n to find p and q.

Factoring a big number like n is hard. The best technique to factor 512 bit number takes 30,000 MIPS-years!

Efficiency of RSA Operations:

Exponentiation

How to compute 123^{54} mod 678?

$123^2 = 123.123 = 15129 = 213 \bmod 678$
$123^3 = 123.213 = 26199 = 435 \bmod 678$
$123^4 = 123.435 = 53505 = 621 \bmod 678$
......
$123^{54} =$ ...... $= 87 \bmod 678$

This requires 54 small number **multiplications**
and 54 small number **divisions**.

How to compute 123^{32} mod 678?

123^2 = 123.123 = 15129 = 213 mod 678
123^4 = 213.213 = 45369 = 621 mod 678
123^8 = 621.621 = 385641 = 537 mod 678
123^{16} = 537.537 = 288369 = 219 mod 678
123^{32} = 219.219 = 47961 = 501 mod 678

This requires **5 multiplications**
and 5 **divisions** instead of 32.

To efficiently compute 123^{54}: 54 is represented in binary as:

1 1 0 1 1 0

$(((((123^2)123)^2)^2 123)^2 123)^2$

This requires **8 multiplications** and 8 **divisions** instead of 32.

Each 1 requires two multipliactions and two divisions
and each 0 requires one multipliaction and one division.

Thus in the above we have three 1s and two 0s
and that yields: 3.2 + 2.1 = 8.
Note that we ignore the leading 1.

Another example: y^{14}, 14 is represented in binary as:

1 1 1 0

$(((y^2) y)^2 y)^2$

This requires **5** multiplications and 5 divisions instead of 32.

Generating RSA Keys

Finding e:

Two popular values for e are: 3 and 65537 ($2^{16} + 1$).

These make public key operations on message m faster (encryption and signature verification is m^e):

- m^3 requires 2 multiplications & 2 divisions.
- m^{65537} requires 17 multiplactions & 17 divisions since the binary value of 65537 is 100..01 (15 zeros).

Finding n:

If e = 3:

Choose random numbers x and y then:

p = (2x+1)*3+2 &

q = (2y+1)*3+2

If e = 65537:

Randomly choose p and q and make sure that they are not 1 mod 65537 (the probability of rejection is 1 in 2^{16}).

Once we selected p and q, then:

$n = p.q$ &

$\varnothing(n) = (p-1)(q-1)$.

Finding d:

How to fine d such that $e.d = 1 \bmod \varnothing(n)$?
Use ***Euclid*** algorithm (see Section 7.4, page 187 of textbook).

The RSA keys:

public key: <3 | 65537, n> **private key: <d , n>**.

Diffie-Hellman

Alice and Bob agree on: p (large prime) & g < p.

Alice

Pick S_A (512-bit random number)

Compute $T_A = (gS_A) \bmod p$

T_A >>>

Compute $X = T_B{}^S A \bmod p$

Bob

Pick S_B (512-bit random number)

Compute $T_B = (gS_B) \bmod p$

<<< T_B

Compute $Y = T_A{}^S B \bmod p$

X is the same as Y!

why?

$X = T_B^{SA} = g^{SBSA}$
$Y = T_A^{SB} = g^{SASB}$

No one can compute $g^{(SASB)}$ by knowing $g^{(S_A)}$ & $g^{(S_B)}$

The bucket Brigade/Man-in-theMiddle Attach

Alice	Mr. X	Bob
Pick S_A	Pick S_X	Pick S_B

Compute: $T_A = g^{S_A} \bmod p$ $\quad T_X = g^{S_X} \bmod p$ $\quad T_B = g^{S_B} \bmod p$

T_A	>>	$T_A \,..\, T_X$	>>	T_X
T_X	<<	$T_X \,..\, T_B$	<<	T_B

Compute: $K_{AX} = T_X^{S_A} \bmod p$ $\quad K_{AX} = T_A^{S_X} \bmod p$ $\quad K_{BX} = T_X^{S_B} \bmod p$

$K_{BX} = T_B^{S_X} \bmod p$

Possible Defense

- **Each person *i* picks S_i and computes $T_i = g^{S_i} \bmod p$ *and Keeps S_i private and makes T_i public***
- **If Alice like to communicate with Bob, she finds T_B and computes:**

  $$K_{AB} = T_B^{S_A} \bmod p$$

- **Then tells Bob she likes to communicate with him.**
- **Bob finds T_A and hen computes:**

  $$K_{BA} = T_A^{S_B} \bmod p$$

This requires PKI (*public Key Infrastructure*) to manage T_i

ElGammal Signature

- **Each person has *long-term* *public*/*private* key pair:**

  Private: S
  Public: $< g, p, T >$ **where** $T = gS \bmod p$

- **For each message m to be signed:**

  - **Generate a *per-message* *public*/*private* key pair:**

    Private: S_m
    Public: $T_m = g^S m \bmod p$

  - **Compute the message digest:** $d_m = \mathrm{MD}(\mathrm{m} \mid T_m)$

  - **Compute the message signature:** $X = S_m + d_m{}^S \bmod (p - 1)$

  - **Send:** m, T_m **and** X

Verify:

- **Compute the message digest:** $d_m = \mathrm{MD}(\mathrm{m} \mid T_m)$

- **Compute:** $\mathrm{Y1} = g^X$ **and** $\mathrm{Y2} = T_m T^d m$

  and If $\mathrm{Y1} = \mathrm{Y2}$ then the *signature is correct*.

Why?

$\mathrm{Y1} = g^X = g^S m^{+dmS} = g^S m \; g^{dmS} = T_m g^{S\,dm} = T_m T^d m = \mathrm{Y2}$

Digital Signature Standard (DSS)

Proposed by NIST based on a modified version of ElGammal algorithm.

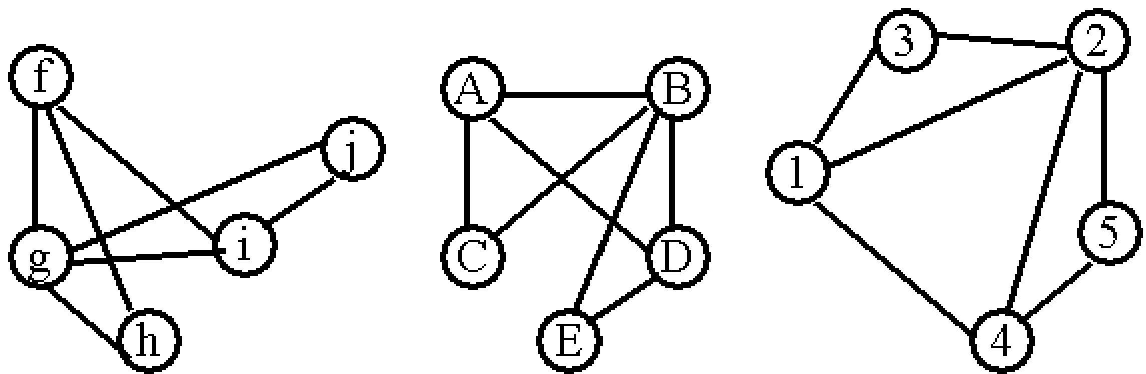
Zero Knowledge Proof Systems

Example:

***Graph isomorphism problem*:**

We consider two graphs *isomorphic* if we can *rename* the vertices of one to get a graph *identical* to the other.

This is a well known NP-complete problem.



f > A > 1
g > B > 2
h > C > 3
i > D > 4
j > E > 5

Alice specifies a large graph G_A and renames the vertices to produce another isomorphic graph G_B.

***Public Key:* (G_A , G_B)**
***Private Key:* $G_A \leftrightarrow G_B$**

To prove to Bob that she is Alice:

- **She renames the vertices to produce a set of isomorphic graphs: G_1 G_2 .... G_k and sends them to Bob.**

- **Bob asks Alice to show him for each i the mapping between: G_i and *either* G_A or G_B but *not both,* otherwise Bob may knows her private key!**