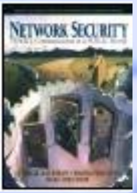
Data Protection,
Chapter 5

Authentication and Digital Signatures

Chapter Index

- **Hash functions**
  - MDC: Modification Detection Codes
  - MAC: Message Authentication Codes
- **Digital Signatures**
  - RSA, ElGamal
  - DSS: Digital Signature Standard
- **Certificates (name and attribute)**
  - X.509
- **Key distribution mechanisms in asymmetric cryptography**

Bibliography



- **Basic:**
  - Kaufman[5,7],
  - Stallings [10,11,12,13],
- **Complementary:**
  - Ramió [15,16,17]
  - Lucena [13,17]
  - Menezes [9,11,12,13]

Complementary bibliography

- FIPS PUB 113, *Computer Data Authentication*, NIST, May 1985. *Available at:* *http://www.itl.nist.gov/fipspubs/fip113.htm*
- *FIPS PUB 180-1, Secure Hash Standard, NIST, April 1995. Available at :* *http://www.itl.nist.gov/fipspubs/fip180-1.htm*
- FIPS PUB 186, *Digital Signature Standard*, NIST, May 1994. *Available at* : http://www.itl.nist.gov/fipspubs/fip186.htm
- R.L. Rivest, *RFC 1321: The MD5 Message-Digest Algorithm*, Internet Activities Board, 1992. *Available at* : http://www.ietf.org/rfc/rfc1321.txt?number=1321

Motivation

Integrity: Problems I

a) **Message integrity:**

- ❖ How can Bob validate that the message received from Alice is authentic, i.e. has not been fabricated nor changed during transmission?

b) **Data origin authentication:**

- ❖ How can Bob validate that a message received from a sender, who claims to be Alice, actually is coming from Alice?
  - ✓ Dealing with authenticity of the sender
  - ✓ Usually includes message integrity

Integrity: Problems II

c) **Non-repudiation of the sender:**

- How can Bob validate that a message sent from Alice actually has been sent by Alice, even if Alice claims not having sent the message?

d) **Non-repudiation of the receiver:**

- How can Bob validate that a message sent to Alice actually has arrived at Alice even if Alice claims not having received the message?

e) **Impersonation of the identity of the sender / the receiver**

- How can Bob check if Alice, Clare, or other users are sending messages as if they were signed by Bob?

How do we achieve that?

- The following slides show how to achieve the previous objectives taking into account both the techniques already studied and those introduced in this chapter

Sender Authentication

- **Two basic methods:**
  - Symmetric cryptography using shared secret:
    - ✓ Explicit exchange (seen)
    - ✓ Challenge/Response (seen)
    - ✓ Symmetric encoding with CRC (seen)
    - ✓ **Use of hash functions with a secret key (this chapter)**
  - Asymmetric cryptography using digital signatures
    - ✓ This chapter

Message Integrity

- **Two basic alternatives:**
  - Symmetric encoding with CRC
    - ✓ Symmetric key does not need to be a shared secret
    - ✓ If the message has redundancy or the padding helps to detect modifications, the CRC would not be needed
  - MDCs (modification detector code) using a key or coded (this chapter)
- **Other alternatives may involve the use of secure channels**
  - A not very common case

Non-repudiation and Impersonation

- **Non-repudiation:**
  - Digital signatures (this chapter)
- **Impersonation:**
  - Digital certificates (this chapter)

UNIVERSIDAD CARLOS III DE MADRID

homo homini sacra res

Hash Functions

Departamento de INGENIERIA TELEMATICA www.it.uc3m.es

Hash Functions

- **Hash functions (also known as message digest):**
  - Input: A message with an arbitrary length
  - Output: A fingerprint / marker / digest / hash value / ...
    - ✓ Having a fixed length (**`n`** bits)
- **Calculating a hash function can be done:**
  - From the message alone: MDC
  - Using the message and an additional key: MAC
- **Two principal applications in cryptography:**
  - Assure **integrity** for a message
  - Provide **authentication and integrity** for a message

Hash functions v.s. CRC

- **Hashes are just a fingerprint of fixed length, so isn't this the same as a CRC?**
- **The answer is NO:**
  - CRC was designed to countermeasure noise:
    - ✓ A CRC allows the extraction of information of the original message
    - ✓ It is easy to obtain several messages with the same CRC
  - Hash functions are designed as protection against **malicious** attackers

Desired Properties of Hash Functions

- **Unidirectional (Pre-image resistance):**
  - Computationally infeasible to find a message which results in a pre-specified hash value
- **Compression**
  - A message of any length must have a digest of fixed length.
- **Easy computation**
- **Diffusion**
  - The digest must be a complex function of all the bits of the message. If only one bit is modified, the digest should flip almost half of its bits

Desired Properties of Hash Functions

- **Simple collision (2nd pre-image resistance):**
  - ❖ Computationally infeasible to find one message which results in the same hash value as a pre-specified message
- **Strong collision resistance:**
  - ❖ Computationally infeasible to find any two messages which result in the same hash value
  - ❖ Note: Requires less operations for brute-force than the former two properties
- **Remember birthday paradox:**
  - ❖ You do not need to search within a 2^m space of messages, searching within a $2^{m/2}$ will suffice.
    - ✓ Algorithmic complexity drastically reduced.

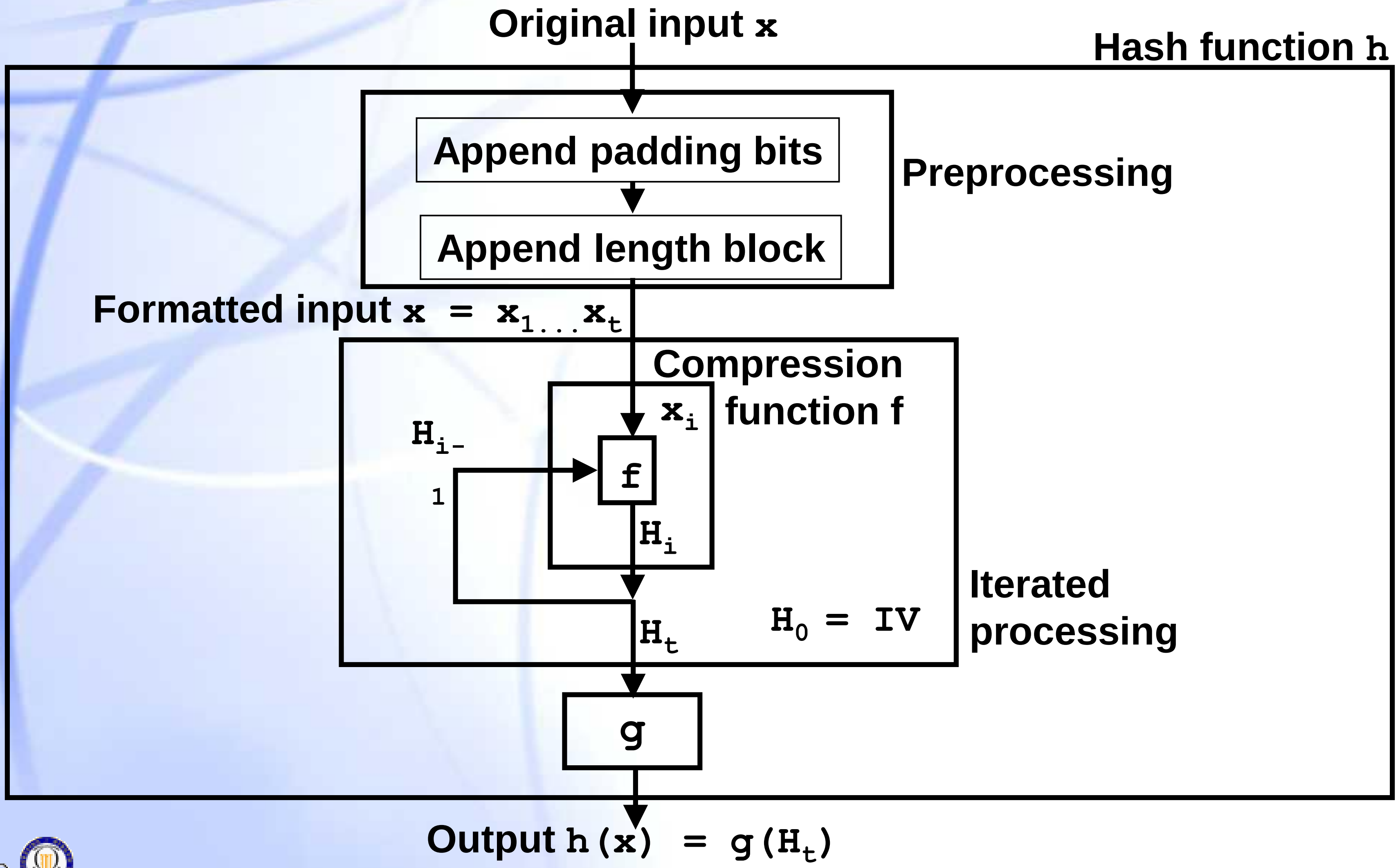
Review of the birthday paradox

- Determine how many people is needed in a room so that al least two of them have the same birthday with probability greater than 0,5.
- Actually, this is not a paradox but it seems so because the number of people needed is 23 only ($366^{1/2}$ = 19)
- Explanation: if each person enters the room and deletes from the blackboard his/her birthday, this first will have a probability that his/her birthday is not already deleted of n/n = 1, the second of (n-1)/n, etc. The probability of non-coincidence is pNC = $n!/(n-k)!n^k$. If k = 23, we have that pNC = 0,493 and the probability of coincidence is pC = 0,507.

Types of hash functions

- **MDC (manipulation detection codes) or MIC (message integrity codes) → do not use keys**
  - Unidirectionals or One-Way Hash Functions (OWHFs)
  - Collision resistant (strong) o Collision Resistant Hash Functions (CRHFs)
- **MAC (message authentication codes) → use keys**
  - The implicit key provides integrity, and if the key is a shared secret, it also provides authentication.

Generic Model for Iterated Hash Functions



Merkle's Meta-Method for Hashing

- **INPUT: compression function `f` which is collision resistant**
- **OUTPUT: unkeyed hash function `h` which is collision resistant**
  - Suppose that **f** maps (**n+r**) bits to **n**-bit outputs
  - Break an input **x** into **t** blocks $\mathbf{x_i}$ **(i=1,2,...,t)**
    - All having the block length **r**
    - Pad the last block with zeros if necessary
  - Add an additional block with the message length
  - Letting $\mathbf{H_0} =$ **'0'**$^{\mathbf{n}}$ (i.e. **n** zeros), compute iteratively the hash from:
    - $\mathbf{H_i = f(H_{i-1} \mid\mid x_i)}$ and append the length block

Unkeyed Hash Functions:
MDCs Based on Block Ciphers

- Most famous examples:

Hash function	n	m	Preimage	Collision	Comments
Matyas-Meyer-Oseas	n	n	2^n	$2^{n/2}$	For key length=n
MDC-2 (with DES)	64	128	$2*2^{82}$	$2*2^{54}$	Rate = 0.5
MDC-4 (with DES)	64	128	2^{109}	$4*2^{54}$	Rate = 0.25
MD4	512	128	2^{128}	2^{20}	
MD5	512	128	2^{128}	Also broken	
RIPEMD-128	512	128	2^{128}	?	
SHA-1, RIPEMD-160	512	160	2^{160}	2^{63}	

MDCs Based on Block Ciphers

- **Motivation:**
  - ❖ Efficient block ciphers already wide-spread
  - ➢ Construct a hash function from a block cipher
- **But:**
  - ❖ Block ciphers do not possess the properties of random functions which would be ideal to build hash functions
    - ✓ E.g., they are invertible…
- **Define the rate of an MDC as the inverse of the number of block cipher iterations it uses**

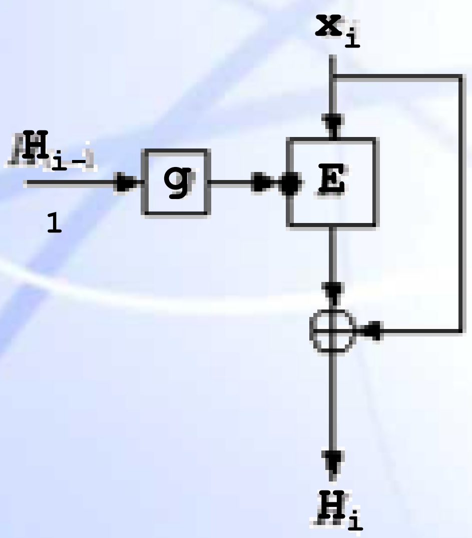
MDCs Based on Block Ciphers: Example Rates

- n: n-bit block ciphers
- k: size of the block cipher key (in bits)
- m: size of the hash value (in bits)

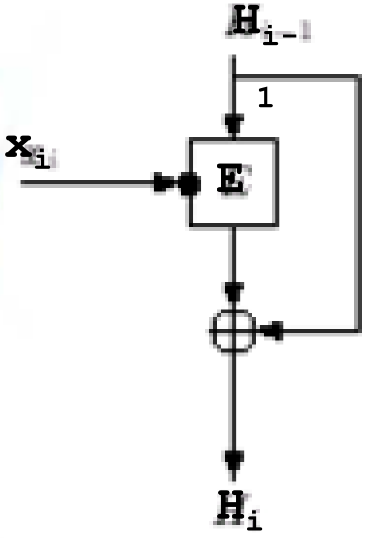
Hash function	*(n, k, m)*	*Rate*
Matyas-Meyer-Oseas	(n, k, n)	1
Davies-Meyer	(n, k, n)	k/n
Miyaguchi-Preneel	(n, k, n)	1
MDC-2 (with DES)	(64, 56, 128)	½
MDC-4 (with DES)	(64, 56, 128)	¼

MDCs of Rate 1

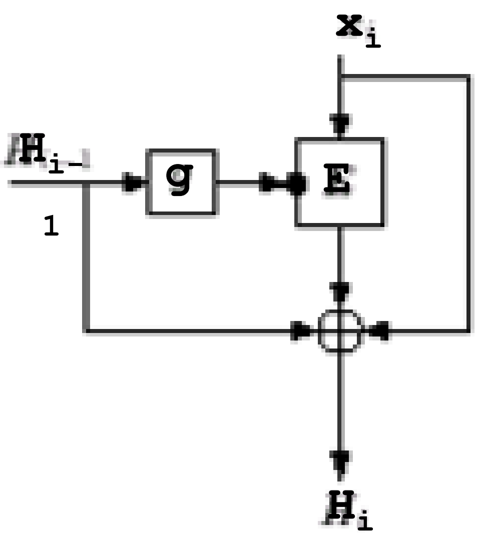
- Based on one iteration of a block cipher
- The main ones are:



Matyas-Meyer-Oseas



Davies-Meyer



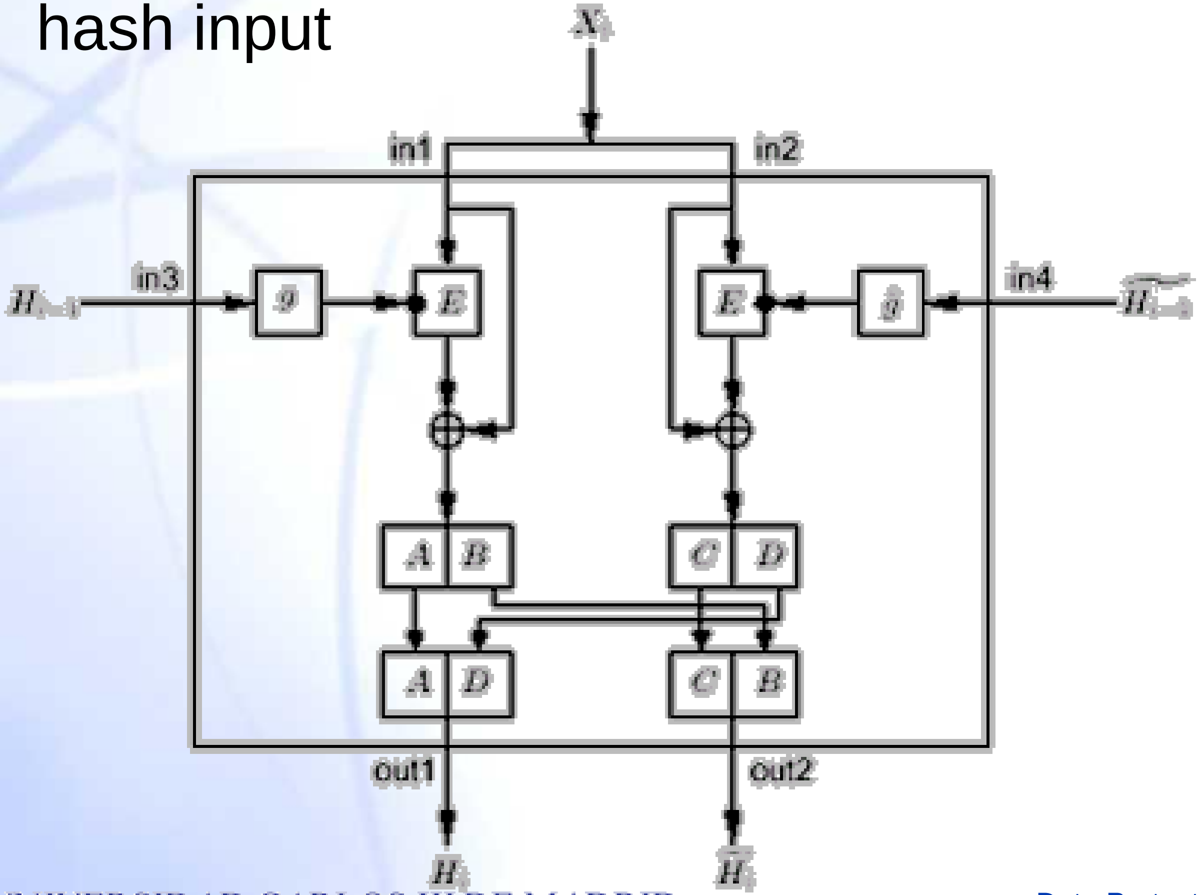
Myaguchi-Preneel

Example: Matyas-Meyer-Oseas Hash

- **INPUT: bit string x**
- **OUTPUT: n-bit hash-code of x**
- **Algorithm:**
  - ❑ Divide input x into n-bit blocks (pad if necessary)
    - ❖ $\mathbf{x_i}$ **(i=1,2,...,t)**
    - ❖ Pre-specify an **n**-bit initialization vector IV
  - ❑ Define a function g that can generate a valid key from $\mathbf{H_i}$.
  - ❑ The output is $\mathbf{H_i = E_{g(H_{i-1})}(x_i) \oplus x_i,\ 1 \le i \le t}$
    - ❖ With: $\mathbf{H_0 = IV}$

MDCs of Rate 0.5: MDC-2

- **MDC-2 algorithm overview:**
  - Uses two block cipher operations per block of hash input

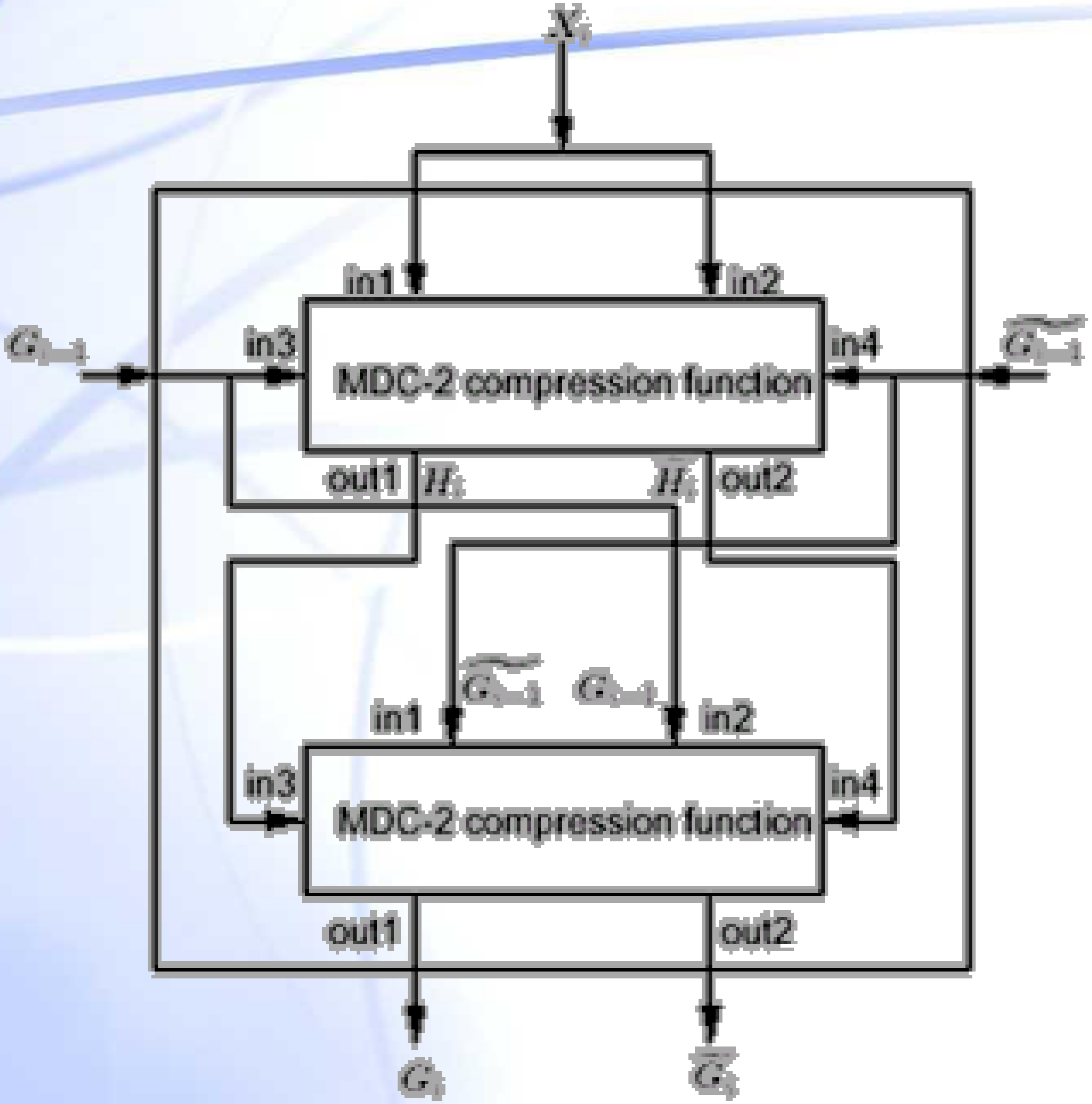


More Details on MDC-2

Components

- DES as underlying block cipher
- Two functions to map 64 bit to 56 bits:
  - $g(U) = u_1\ 1\ 0\ u_4\ u_5\ u_6\ u_7\ u_9\ u_{10}\ .\ .\ .\ u_{56}$
  - $\tilde{g}(U) = u_1\ 0\ 1\ u_4\ u_5\ u_6\ u_7\ u_9\ u_{10}\ .\ .\ .\ u_{56}$
  - **This is used as key input for DES**
    - **These are guaranteed not to be weak keys**
- Definition of the initialization vector:
  - $IV = 0x5252525252525252$
  - $\widetilde{IV} = 0x2525252525252525$
  - $H_0 = IV\ ;\ \tilde{H}_0 = \widetilde{IV}$

MDC of Rate 0.25: MDC-4



Customized MDC Hash Functions

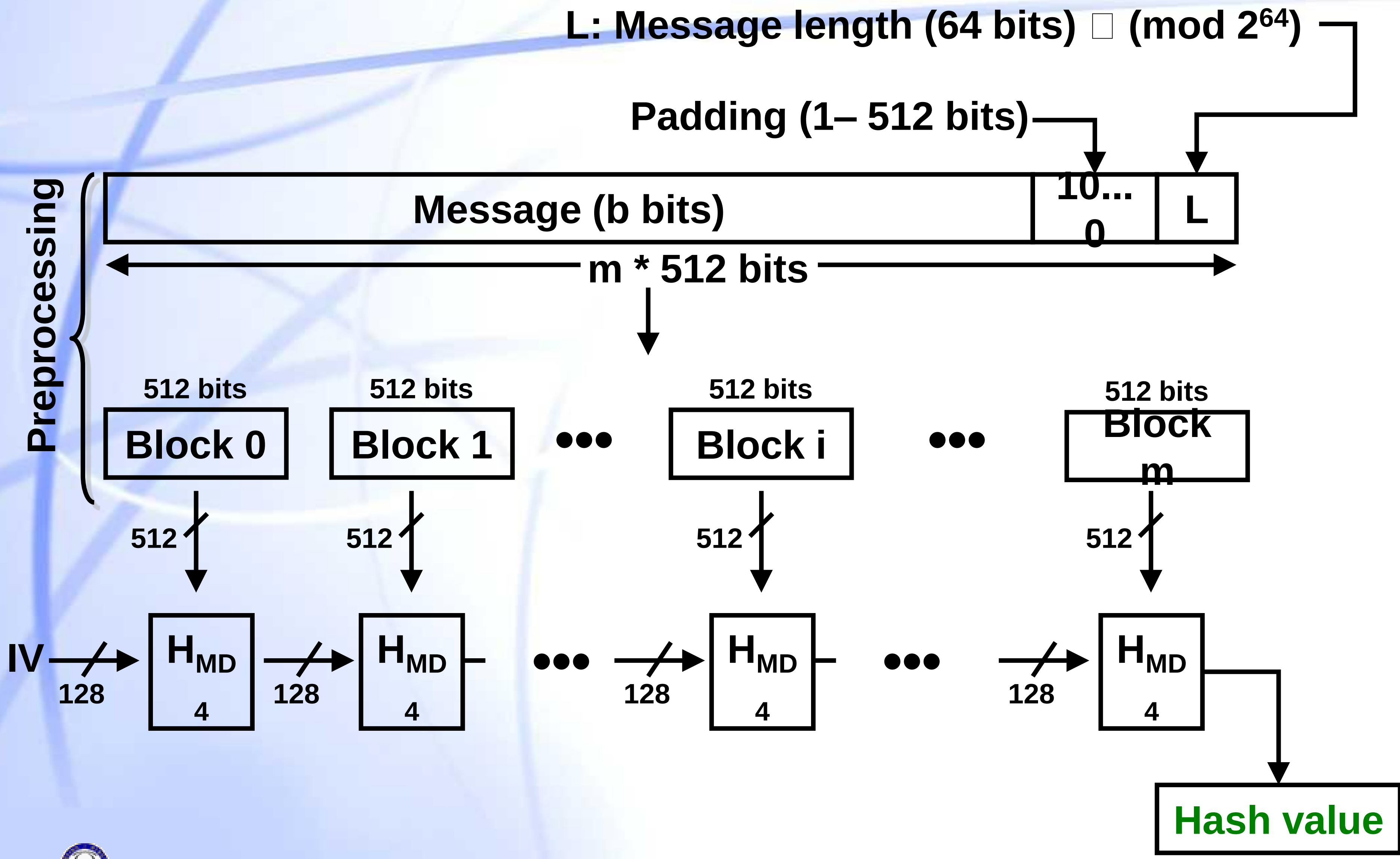
- **Some of the most important ones:**
  - MD4 (message digest 4)
  - MD5 (message digest 5)
  - SHA-1 (Secure hash algorithm 1)
- **MD-5 is an improvement over MD-4**
- **SHA-1 is another improvement**
  - Predecessor SHA-0, published by NIST in 1993, rendered obsolete by vulnerabilities
  - Successor: SHA-224, SHA-256, SHA-384, y SHA-512
- **Other functions of this type:**
  - RIPEMD
  - RIPEMD-128
  - RIPEMD-160

MD4: Message Digest Number 4

- **128-bit hash function**
- **Original goals: Make it difficult to**
  1. Find two messages with the same hash-value:
     - ✓ 2^{64} operations (collision resistance)
  2. Find a message for a pre-specified hash:
     - ✓ 2^{128} operations (2^{nd} pre-image resistance)
- **Problem:**
  - ❖ Goal 1 was missed:
    - ✓ 2^{20} operations necessary only

➢ **MD4 no longer recommended!**

MD4: Operation Overview



L: Message length (64 bits) □ (mod 2^{64})

Padding (1– 512 bits)

Preprocessing

Message (b bits)

10...0

L

m * 512 bits

512 bits

Block 0

512 bits

Block 1

●●●

512 bits

Block i

●●●

512 bits

Block m

512

512

512

512

IV

128

H_{MD4}

128

H_{MD4}

●●●

128

H_{MD4}

●●●

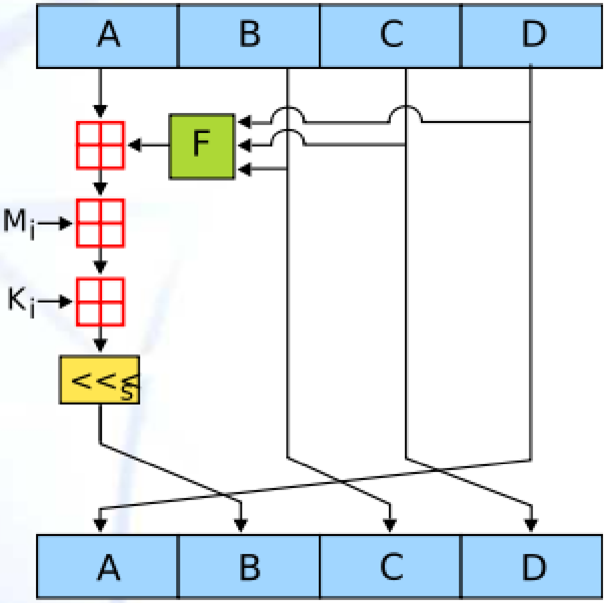
128

H_{MD4}

Hash value

MD4 estructure

H_{MD4} → repeat 48 times (16*3)



Security considerations

- **SHA-1 was compromised in 2005.**
  - MD5 in 2004
- **Xiaoyun Wang, Yiqun Lisa Yin & Hongbo Yu (the same which broke MD-5 the year before) broke SHA-1 with at least 2^{69} operations, (by brute force 2^{80} operations).**
- **Last attacks on SHA-1 have reduced the number of operations to 2^{63}.**
- **Although 2^{63} is a large number of operations, it is within the limit of current computation capabilities.**

Hash Functions Based on Modular Arithmetic

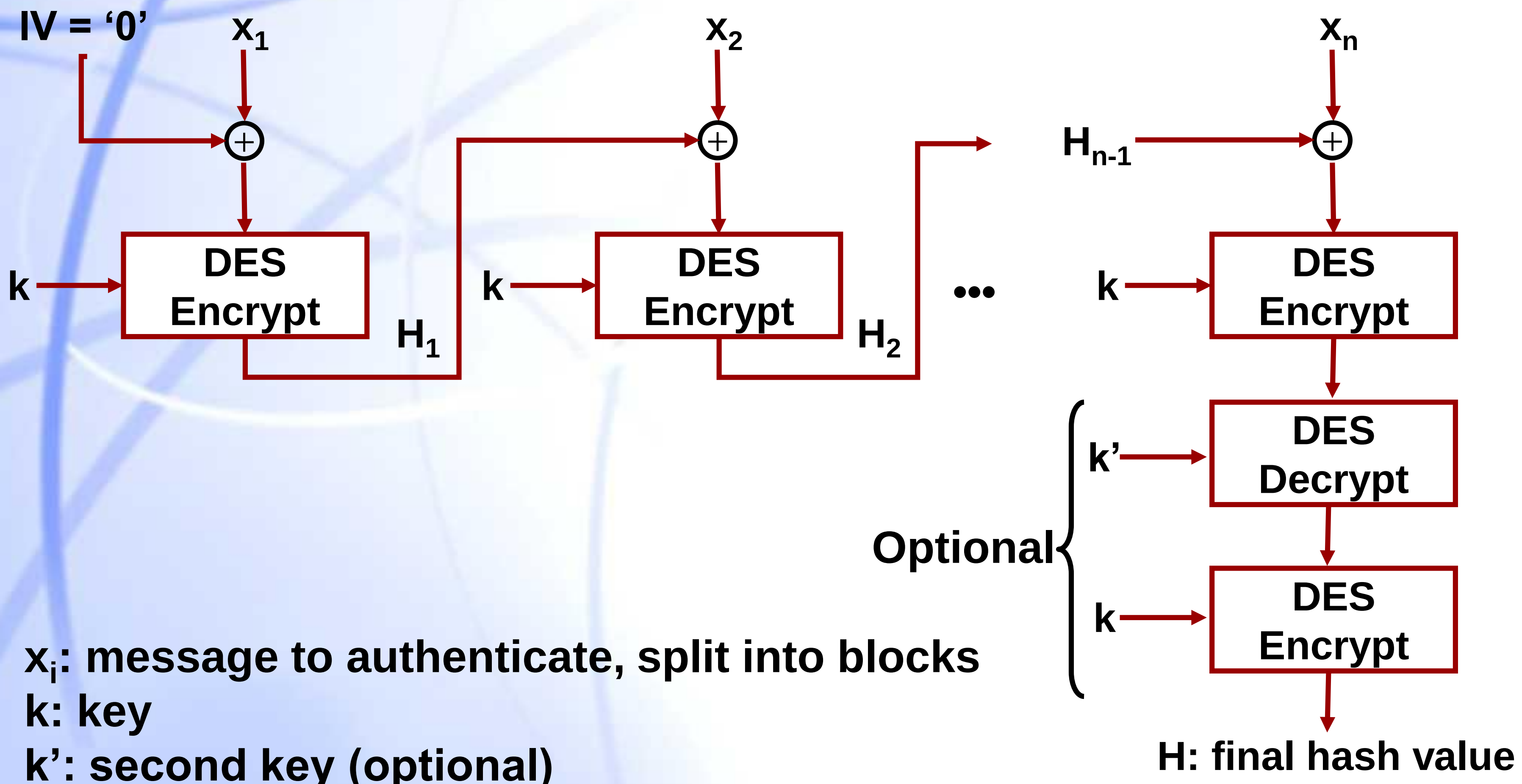
- Use arithmetic modulo m as principle for the compression function
- May re-use existing software based on public-key technology (e.g., RSA)
- Two significant disadvantages:
  - Processing speed
  - Guarantee of security
- One example: MASH

Message Authentication Codes (MAC)

- **Hash function using a key k to provide authentication**
- **Can be computed:**
  - Based on block ciphers
  - Starting from MDCs and adding a key to the input message

MACs Based on Block Ciphers

- Example: CBC-MAC algorithm:



x_i: message to authenticate, split into blocks

k: key

k': second key (optional)

H: final hash value

Constructing MACs from MDCs I

- **Example: Keyed-Hashing for Message Authentication Codes (HMAC)**
- **H. Krawczyk, M. Bellare, R. Canetti, RFC 2104, 1997.**
- **Define:**
  - H: Generic hash function (MD5, SHA-1...)
  - K: Secret key shared between two peers
  - B: Block length for the input of H (in bytes)
  - L: Length of the output of H (in bytes)
    - ✓ MD5: $L = 16$ (128 bit); SHA-1: $L = 20$ (160 bit)
  - K_L: Length of the key K
- **Recommendation: K_L should be at least L**
- **If $K_L > B$: obtain a new $K' = H(K)$**

Constructing MACs from MDCs II

- **Procedure to obtain the hash value of `M`:**
  - Pad `K` or `K'` with ‘0’s to create a string `K”` with `B` bytes
  - Define:
    - ✓ `ipad` = the byte 0x36 repeated `B` times
    - ✓ `opad` = the byte 0x5C repeated `B` times
  - To compute the hash value `S` for message `M`:
    - ✓ `S = H(K” XOR opad || H(K” XOR ipad || M))`
- **Example:**
  - H = MD5 to obtain a keyed variant of MD5:
    - ✓ Used in IPsec

Digital Signatures

What are we going to study?

- **Digital signature requirements**
- **Properties**
- **Methods**
  - RSA
  - ElGamal
  - DSS

What do we want?

- **Obtain the following security services:**
  - Integrity
  - Authentication
  - Non-repudiation
- **How?**
  - By means of digital signatures

Characteristics of a Digital Signature

- **Requirements:**
  - Should be easy to generate
  - Should be irrevocable, not to be rejected by its owner
  - Should be unique (only the owner can generate it)
  - Should be easy to authenticate or recognize by its owner and the receivers
  - Must depend on the message and the author

Much stronger requirements than for a hand-written signature!

Digital signature

- **We need to leave a print that only the sender could leave in a message**
  - ❖ Use of the private key
  - ❖ On the whole message?
    - ✓ Only on the digest (speed)
- **Review 3 systems:**
  - ❖ RSA
  - ❖ ElGamal
  - ❖ DSS

Digital Signatures using RSA

Public key (n_A, e_A) Private key (d_A)

Alice

Algorithm:

Signature: $s_A[H(M)] = H(M)^{dA} \bmod n_A$

A sends a message M (plaintext or ciphertext) to destination B together with the signature: $\{M,\ s_A[H(M)]\}$

Bob

B has the public key (n_A, e_A) of A and decrypts $s_A[H(M)] \Rightarrow \{(H(M)^{dA})^{eA} \bmod n_A\}$ to get $H(M)$.
On reception of message M', B computes the hash function $H(M')$ and compares:

If $H(M') = H(M)$ the signature is correct. ✌

Digital Signatures using RSA: Example

Bob

Hi! I will send you a document. Regards, Bob.

Alice

Assume: **H(M) = F3A9** (16 bits)

Bob's keys:
$n_B = 65.669$
$e_B = 35,\ d_B = 53.771$

$2^{16} \bullet 65.669 \bullet 2^{17}$
Then, sign the 16-bit block hash using a 17-bit signature

Alice keys
$n_A = 66.331$
$e_A = 25,\ d_A = 18.377$

The prime numbers (p,q) chosen by Bob are (97, 677)

Signature:

$H(M) = F3A9_{16} = 62.377_{10}$

$s_{H(M)} = H(M)^{dB} \bmod n_B$

$s_{H(M)} = 62.377^{53.771} \bmod 65.669 = 24.622$

Bob sends the pair (M,s) = (**M, 24.622**)

Digital Signatures using RSA: Validation of the Signature

Bob

Bob's keys:

n_B = 65.669

e_B = 35, d_B = 53.771

Alice's keys

n_A = 66.331

e_A = 25, d_A = 18.377

Alice

From the previous slide: $\mathtt{H(M) = F3A9_{16} = 62.377_{10}}$

$\mathtt{s_{H(M)} = H(M)^{dB} \bmod n_B} \quad \mathtt{s_{H(M)} = 24.622}$

Bob has sent the pair (M, r) = (**M, 24.622**)

Alice receives a message **M′** together with a signature **s = 24.622**:

- Compute $\mathtt{s^{eB} \bmod n_B = 24.622^{35} \bmod 65.669 = 62.377}$
- Compute the hash value of **M′**, i.e. **H(M′)**, and compare with **H(M)**.
- If the messages **M** and **M′** are the same, then **H(M) = H(M′)** and the signature is accepted as valid
- **NOTE**: However, **H(M) = H(M′)** does not imply that **M = M′** !

Possible vulnerability of RSA signature

- Working in a multiplicative body, the following scenario is possible:
- If the signature of two messages M_1 and M_2 is known, a third message can be signed M_3 which is a product of the two previous messages, without the need for knowing the private key of the signer.
- Let the signer keys be e, d & n = p∗q. Therefore:
- $r_{M1} = M_1^d \bmod n$ & $r_{M2} = M_2^d \bmod n$ If now $M_3 = M_1 M_2$:
- $r_{M3} = (M_1 M_2)^d \bmod n = M_1^d M_2^d \bmod n = r_{M1}\, r_{M2} \bmod n$

- In practice, this is not a problem, why?
  - Before signing, a hash function is applied

Digital Signatures using ElGamal A to B

Alice

ElGamal: User **A** generates a random number $\mathbf{x_a}$ (the private key) in a field with modulus $\mathbf{p}$. The public key is $\mathbf{y_a = g^{xa} \bmod p}$, with the generator $\mathbf{g}$

Digital signature algorithm

Signature: $(\mathbf{r}, \mathbf{s})$

1. **A** generates a random number $\mathbf{h}$, which should be coprime to $\mathbf{\varphi(p)}$, i.e.: $\mathbf{h \mid \gcd\{h,\varphi(p)\} = 1}$
2. Compute: $\mathbf{h^{-1} = inv\{h,\varphi(p)\}}$
3. Compute $\mathbf{r = g^h \bmod p}$
4. Solve the following congruence to get $\mathbf{s}$:

$$\mathbf{H(M) = x_a*r + h*s \bmod \varphi(p) \quad \Leftrightarrow}$$
$$\mathbf{s = (H(M) - x_a*r) * inv[h,\varphi(p)] \bmod \varphi(p)}$$

Digital Signatures using ElGamal: Validation by B

Bob

Knows:

p and $y = (g^{xa}) \bmod p$

Algorithm for verifying the signature, B:

1. receives the pair (r, s) and computes:
   - $r^s \bmod p$ and $y^r \bmod p$
2. Computes $k = [y^r * r^s] \bmod p$
   - Since $r = g^h \bmod p$ and $y = g^{xa} \bmod p$:
3. $k = [(g^{xar} \cdot g^{hs}] \bmod p = g^{(xar + hs)} \bmod p$
   $= g^{\beta} \bmod p$
4. As H(M) $= (x_a * r + h * s) \bmod \varphi(p)$ and **g** is a simple root of **p**, it is true that:
   $g^{\beta} = g^{\gamma}$ **iff** $\beta = \gamma \bmod (p-1$
5. Verify that $k = g^{H(M)} \bmod p$

If: $k = [(g^{xa})^r * r^s] \bmod p$

is the same as $g^{H(M)} \bmod p$

...

Accepts the signature

Digital Signatures using ElGamal: Example

Bob

Hi again! I'm Bob ... See you.

Let **H(M) = A69B** (16 bits)

Alice

Bob's keys

p = 79.903 g = 10

x_b = 20, g^{xb} mod p = 3.631

h = 31

2^{16} • 79.903 • 2^{17}

therefore, sign a 16 bit block with a 17 bit signature

Signature:

1. **h^{-1} = inv(h, φ(p)) = inv (31, 79.902) = 5.155**
2. **r = g^h mod p = 10^{31} mod 79.903 = 11.755**
3. **s = [H(M) - x_b*r] * [inv(h,φ(p)] mod φ(p)**
   - **H(M) = $A69B_{16}$ = 42.651_{10}**
4. **s = [42.651 - 20*11.755] * 5.155 mod 79.902 = 68.539**

Thus, the signature is: (r, s) = (**11.755**, **68.539**)

Digital Signatures using ElGamal: Example Validation

Bob

Bob’s keys

p = 79.903 g = 10

x_b = 20, g^{xb} mod p = 3.631

h = 31

Alice

H(M) = A69B = 42.651

Alice receives the pair (r, s) = (11.755, 68.539)

Validation of the signature:

1. $r^s \bmod p = 11.755^{68.539} \bmod 79.903 = 66.404$
2. $(g^{xb})^r \bmod p = 3.631^{11.755} \bmod 79.903 = 12.023$
3. $(g^{xb})^r * r^s \bmod p = (12.023 * 66.404) \bmod 79.903 = \boxed{64.419} = k$
4. $g^{H(M)} \bmod p = 10^{42.651} \bmod 79.903 = \boxed{64.419}$

If equal, accept the signature

The Generator g in the ElGamal Digital Signature

Bob

Bob's keys

$p = 79.903 \quad g = 10$

$x_b = 20, \; g^{xb} \bmod p = 3.631$

$h = 31$

$g = 10$ is a generator of the field $p = 79.903$ because:

$p-1 = 79.902 = 2*3^2*23*193$

$q_1 = 2; \; q_2 = 3; \; q_3 = 23; \; q_4 = 193$

And if: $10^{(p-1)/qi} \bmod p \neq 1$

$10^{39.951} \bmod 79.903 = 79.902$

$10^{26.634} \bmod 79.903 = 71.324$

$10^{3.474} \bmod 79.903 = 2.631$

$10^{414} \bmod 79.903 = 41.829$

For example, if $g = 11$, you get the value 1 for the exponent 39.951 and g cannot be used for a signature. It will be impossible to validate using the equation $k = g^M \bmod p$.

Digital Signature Standard DSS

- **1991: National Institute of Standards and Technology (NIST) proposes DSA, Digital Signature Algorithm, a variant of the algorithms by ElGamal y Schnoor.**
- **1994: DSA is established as standard known as DSS, Digital Signature Standard.**
- **1996: The government of the USA allows to export Clipper 3.11, in which DSS is built in using a hash function of the type SHS, Secure Hash Standard.**
- **Major disadvantage of ElGamal:**
  - Duplication of the message size to send in the pair **(r, s)** in $\mathbf{Z_p}$ and **φ(p)**
- **However, the ElGamal scheme has been chosen for DSS**

Digital Signature Standard DSS

- **Public parameters of the signature:**
  - A large prime number **p** (512 - 1024 bits)
  - A prime number **q** (160 bits) divisor of **p-1**
  - A generator **g** "of the order **q mod p**"
    - Generator of the order q is the root g modulus p such that q is the smallest integer that fullfils:
      - **q** being much smaller than **p**
      - Condition: $g^q \bmod p = 1$
      - So that:
        - For all t: $g^t = g^{t \pmod q} \bmod p$

DSS Signature: Example A → B

Generation of the signature at A

- Public keys at A: prime number p, q and the generator g
- Secret key of the signature: x_a $(1 < x_a < q)$ random
- Public key of the signature: $y = g^{xa} \bmod p$
  - Difficult to compute x_a from y !
- To sign a message $1 < H(M) < q$, the signer chooses a random number h, $1 < h < q$ and computes:
  - $r = (g^h \bmod p) \bmod q$
  - $s = [(H(M) + x_a * r) * \mathrm{inv}(h, q)] \bmod q$
- The digital signature for M is the pair (r, s)

Verification of the Signature at B

Verification of A's signature at B

- B receives a pair (`r, s`)
- Then, B computes
  - `w = inv(s, q)`
  - `u = H(M) * w mod q`
  - `v = r * w mod q`
- Verification of the following equation:
  - $r = (g^u\ y^v \bmod p) \bmod q$
- If yes, the signature is accepted

The size of the signature is smaller than `p`, i.e. less bits than the modulo of the signature, since `q` was chosen by design to be smaller than `p`

DSS Signature: Example B → A

Hi Alice, I'm Bob and I sign with DSS.

Let `H(M) = 1101000 = 104`

Bob

Alice

Bob's keys

$p = 223$ $q = 37$ $g = 17$,

$x_b = 25$, $y = g^{xb} \bmod p = 30$

$h = 12$

$2^8 \bullet p = 223 \bullet 2^7$

Thus: sign 7-bit block with 8 bit signature

Signing:

1. $\mathrm{inv}(h, q) = \mathrm{inv}(12, 37) = 34$
2. $r = (g^h \bmod p) \bmod q = (17^{12} \bmod 223) \bmod 37 = 171 \bmod 37 = 23$
3. $s = [H(M) + x_b * r] * [\mathrm{inv}\ (h,q)] \bmod q = [104+25*23]*34 \bmod 37 = 35$
4. The digital signature of $H(M) = 104$ is: $(r, s) = (23, 35)$
5. Bob transmits to Alice: $(M, r, s) = (M, 23, 35)$

Verification of the Signature at A

Bob

Bob’s keys

p = 223 q = 37 g = 17,
x_b = 25, y = g^{xb} mod p = 30
h = 12

Alice receives:

(M, r, s) = (M, 23, 35)

Alice

equal?

Accepting the signature

Verification of the signature:

1. w = inv(s, q) = inv(35, 37) = 18
2. u = M*w mod q = 104*18 mod 37 = 22
3. v = r*w mod q = 23*18 mod 37 = 7
4. (g^u y^v mod p) mod q = r ?
5. [($17^{22}30^7$) mod 223] mod 37 = 23

And the size is smaller: q = 37 << p = 223 which was a weak point of ElGamal

Key selection in DSS

- **Choose a prime *q* of 160 bits.**
- **Choose a prime *p* of length L bits, such as $p=qz+1$ for an integer *z*, $512 \leq L \leq 1024$, and *L* is divisible by 64.**
  - The modification “FIPS-186-2, change notice 1” specifies that *L* must be only 1024.
- **Choose *h*, where $1 < h < p - 1$ such as $g = h^z \bmod p > 1$. (remember that $z = (p-1) / q$.)**
- **Choose *x* randomly, where $0 < x < q$.**
- **Compute $y = g^x \bmod p$.**
- **The public key will be (*p*, *q*, *g*, *y*). The private will be *x*.**
- **Note that (*p*, *q*, *g*) can be shared among several users.**

UNIVERSIDAD CARLOS III DE MADRID
homo homini sacra res

Certificates

My identity validated by a third party

Departamento de INGENIERIA TELEMATICA
www.it.uc3m.es

Our objective

- **To be guaranteed the identity of somebody without needing to have knowledge about him/her in advance**
- **In the physical world?**
  - ID card, passport
  - That is: a document which contains identity data, "signed" by someone I trust.
- **In the digital world?**
  - Digital certificates.

Our search

- **How can we be sure that a specific public key belongs to a user?**
  - If we are certain that a public key belongs to somebody we trust (Certificate Authority, CA)
  - And the CA signs also other associations between identity-public key → that is a certificate (of identity)

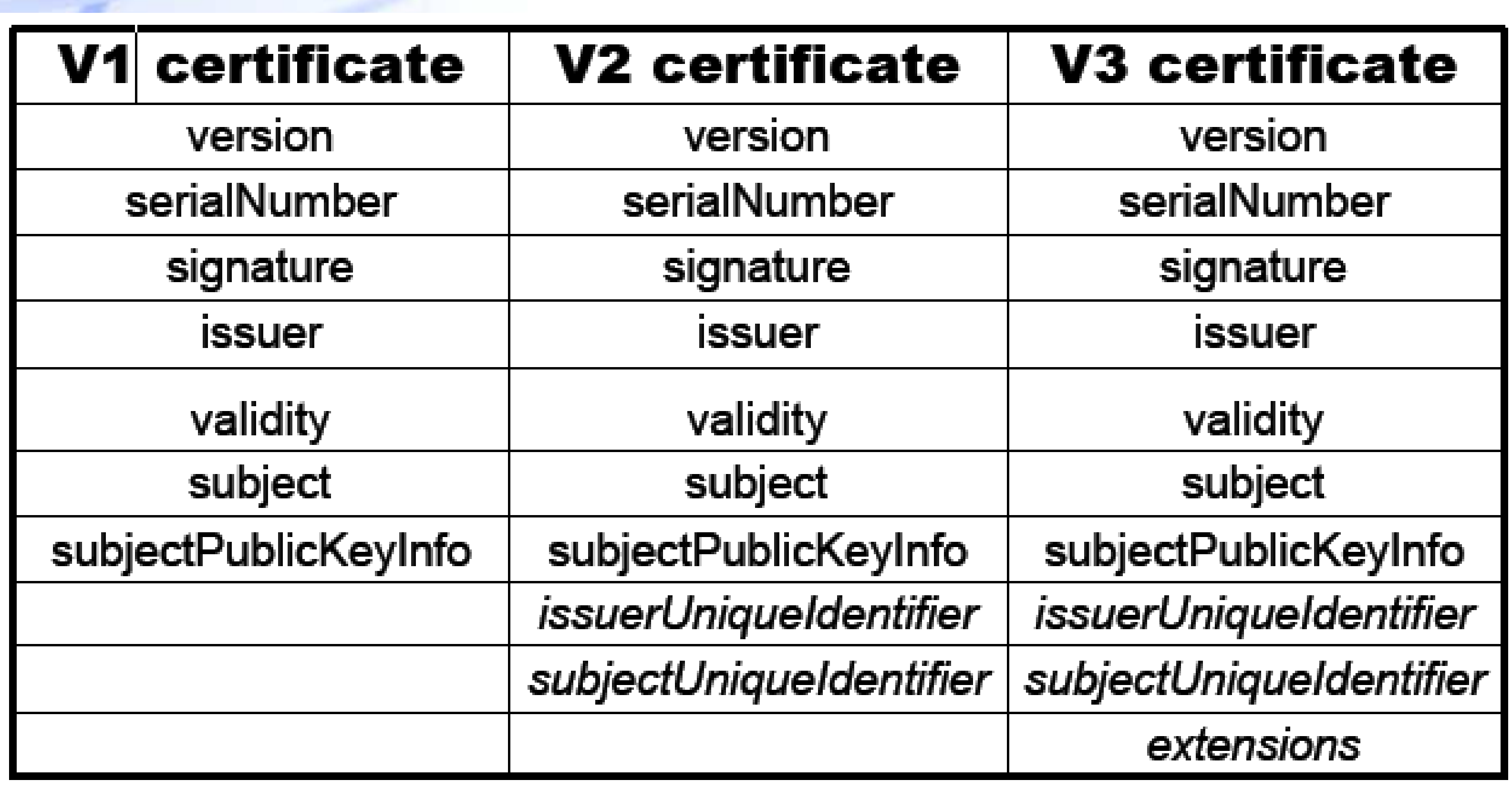
Certificates of identity

- **A certificate consists of:**
  - A public key
  - An identifier of the user,
  - → Signed digitally by a certification authority (CA)
- **Objective:**
  - Show that a public key belongs to a certain user
  - Sender and receiver trust the certificate, so the sender is authenticated from the point of view of the receiver (if the certificate is digitally signed by the CA and the sender proves that it has the private key)
- **Format of certificates:**
  - X.509 (Recommendation X.509 of the CCITT: ”The Directory - Authentication Framework”. 1988)
- **Well-known and currently widely extended**
  - The format X.509 was adopted by PKIX group of IETF and adopted by TCP/IP

Versions

- **ITU-T X.509 standard:**
  - v1 (1988)
  - v2 (1993) = minor changes
  - v3 (1996) = v2 + extensions+ attribute certificates v1
  - v3 (2001) = v3 + attribute certificates v2

Fields of a X.509 certificate



V1 certificate	V2 certificate	V3 certificate
version	version	version
serialNumber	serialNumber	serialNumber
signature	signature	signature
issuer	issuer	issuer
validity	validity	validity
subject	subject	subject
subjectPublicKeyInfo	subjectPublicKeyInfo	subjectPublicKeyInfo
	issuerUniqueIdentifier	*issuerUniqueIdentifier*
	subjectUniqueIdentifier	*subjectUniqueIdentifier*
		extensions

Data format

- **To define a data structure which could be able to travel by a network we need to use a data format definition language (XDR, IDL…).**
- **X.509 makes use of ASN.1**
  - ASN.1 allows the definition of data structures
  - Define coding rules to map those data structures to bits in order to be transmitted from one system to another.

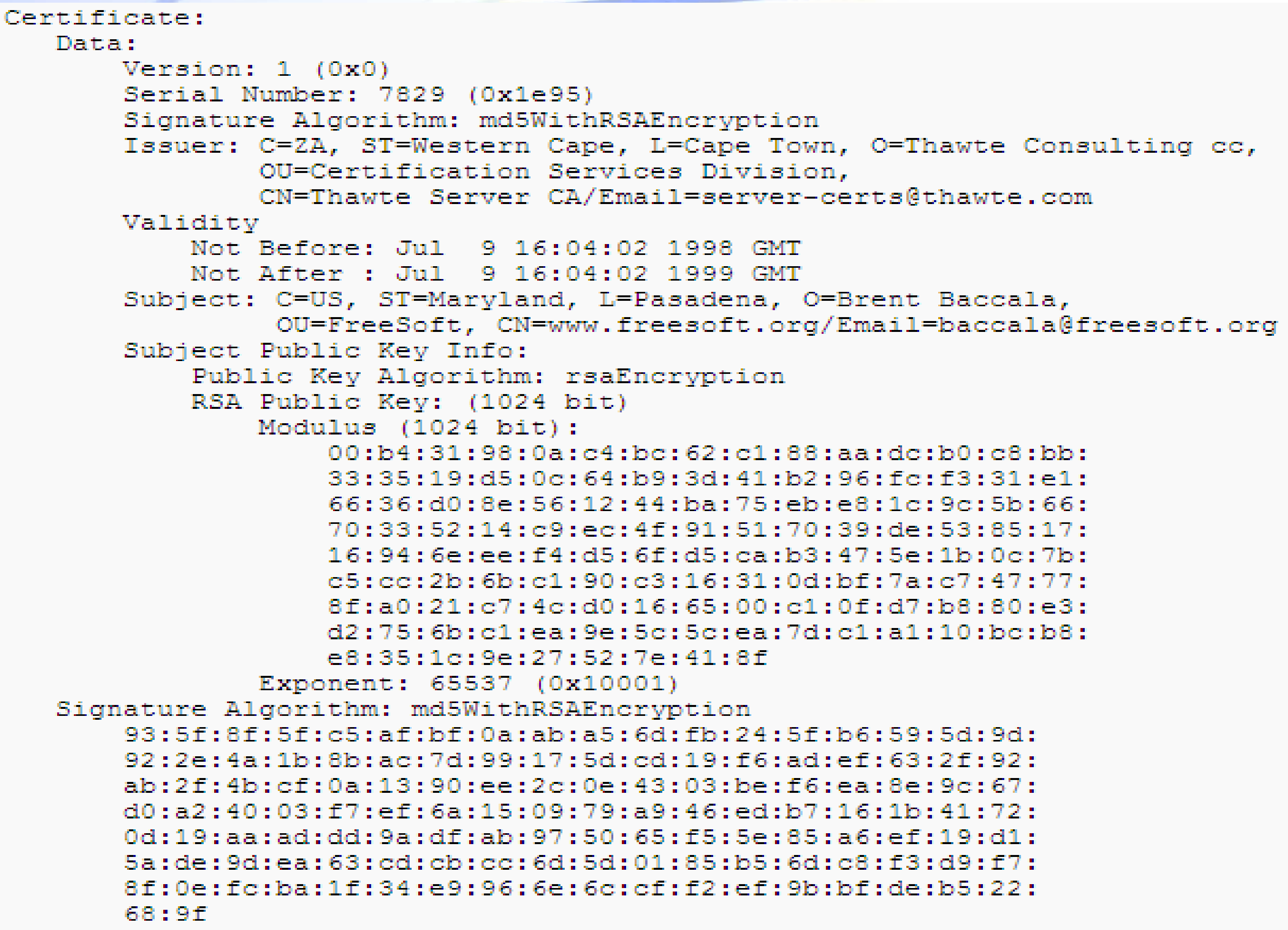
X.509v3 certificates in ASN.1

```
           Certificate  ::=  SEQUENCE  {
          tbsCertificate       TBSCertificate,
        signatureAlgorithm   AlgorithmIdentifier,
           signature            BIT STRING  }

          TBSCertificate  ::=  SEQUENCE  {
        version         [0]  Version DEFAULT v1,
      serialNumber         CertificateSerialNumber,
        signature            AlgorithmIdentifier,
               issuer               Name,
             validity             Validity,
               subject              Name,
       subjectPublicKeyInfo SubjectPublicKeyInfo,
issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                   -- If present, version must be v2 or v3
subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
                   -- If present, version must be v2 or v3
        extensions      [3]  Extensions OPTIONAL
                      -- If present, version must be v3
                            }
```

X.509v3 certificates in ASN.1

```
Version  ::=  INTEGER  {  v1(0), v2(1), v3(2)  }

     CertificateSerialNumber  ::=  INTEGER

          Validity  ::=  SEQUENCE  {
           notBefore           UTCTime,
          notAfter             UTCTime  }

       UniqueIdentifier  ::=  BIT STRING

     SubjectPublicKeyInfo  ::=  SEQUENCE  {
      algorithm              AlgorithmIdentifier,
         subjectPublicKey     BIT STRING  }

     Extensions  ::=  SEQUENCE OF Extension

         Extension  ::=  SEQUENCE  {
          extnID        OBJECT IDENTIFIER,
         critical     BOOLEAN DEFAULT FALSE,
            extnValue   OCTET STRING  }
```

Certificate example X.509



```
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number: 7829 (0x1e95)
        Signature Algorithm: md5WithRSAEncryption
        Issuer: C=ZA, ST=Western Cape, L=Cape Town, O=Thawte Consulting cc,
                OU=Certification Services Division,
                CN=Thawte Server CA/Email=server-certs@thawte.com
        Validity
            Not Before: Jul  9 16:04:02 1998 GMT
            Not After : Jul  9 16:04:02 1999 GMT
        Subject: C=US, ST=Maryland, L=Pasadena, O=Brent Baccala,
                 OU=FreeSoft, CN=www.freesoft.org/Email=baccala@freesoft.org
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:b4:31:98:0a:c4:bc:62:c1:88:aa:dc:b0:c8:bb:
                    33:35:19:d5:0c:64:b9:3d:41:b2:96:fc:f3:31:e1:
                    66:36:d0:8e:56:12:44:ba:75:eb:e8:1c:9c:5b:66:
                    70:33:52:14:c9:ec:4f:91:51:70:39:de:53:85:17:
                    16:94:6e:ee:f4:d5:6f:d5:ca:b3:47:5e:1b:0c:7b:
                    c5:cc:2b:6b:c1:90:c3:16:31:0d:bf:7a:c7:47:77:
                    8f:a0:21:c7:4c:d0:16:65:00:c1:0f:d7:b8:80:e3:
                    d2:75:6b:c1:ea:9e:5c:5c:ea:7d:c1:a1:10:bc:b8:
                    e8:35:1c:9e:27:52:7e:41:8f
                Exponent: 65537 (0x10001)
    Signature Algorithm: md5WithRSAEncryption
        93:5f:8f:5f:c5:af:bf:0a:ab:a5:6d:fb:24:5f:b6:59:5d:9d:
        92:2e:4a:1b:8b:ac:7d:99:17:5d:cd:19:f6:ad:ef:63:2f:92:
        ab:2f:4b:cf:0a:13:90:ee:2c:0e:43:03:be:f6:ea:8e:9c:67:
        d0:a2:40:03:f7:ef:6a:15:09:79:a9:46:ed:b7:16:1b:41:72:
        0d:19:aa:ad:dd:9a:df:ab:97:50:65:f5:5e:85:a6:ef:19:d1:
        5a:de:9d:ea:63:cd:cb:cc:6d:5d:01:85:b5:6d:c8:f3:d9:f7:
        8f:0e:fc:ba:1f:34:e9:96:6e:6c:cf:f2:ef:9b:bf:de:b5:22:
        68:9f
```

X.509v3 extensions

- **Extensions can be:**
  - Public (the same meaning for everybody)
  - Private (particular to a specific community)
- **Can be defined as:**
  - critical – must be understood by the verifier of the certificate
  - non-critical – may be omitted by the verifier

Public extensions

- Information about policies and keys
- Alternative names (non-X.500) for the sender and subject of the certificate
- Restrictions dealing with the path of the certificate
- Identifier of the certificate revocation list where one has to check the legitimacy of the certificate

Information about policies and keys

- **For example, the use of a key:**
  - Sign
  - Encrypt keys
  - Encrypt data
  - Sign certificates
  - Sign certificate revocation lists
- **For example, the identifier of the key of the CA (if it has several keys)**

Alternative names

- **Names can be**
  - E-mail addresses
  - DNS domain names
  - Web URIs (Uniform Resource Identifier)
  - IP addresses
  - X.400 email addresses
  - Registered identifiers
  - Other

Restrictions about the path of the certificate

- **For example:**
  - To specify whether the subject of a certificate is a CA or a final entity → certificate chain
  - To specify the maximum depth of the chain of certificates

Certificate Revocation List Identification

- **Admits several formats:**
  - Entry of a directory
  - E-mail address
  - URL

Certificate Revocation List in X.509

- Signed list which contains the compromised certificates, and therefore, revoked.
- CRL X.509v2

```
        CertificateList  ::=  SEQUENCE  {
         tbsCertList           TBSCertList,
       signatureAlgorithm   AlgorithmIdentifier,
         signature              BIT STRING   }

         TBSCertList  ::=  SEQUENCE  {
       version                  Version OPTIONAL,
                               -- if present, must be v2
      signature                AlgorithmIdentifier,
             issuer                    Name,
            thisUpdate               UTCTime,
            nextUpdate               UTCTime,
     revokedCertificates     SEQUENCE OF SEQUENCE  {
      userCertificate          CertificateSerialNumber,
              revocationDate             UTCTime,
       crlEntryExtensions       Extensions OPTIONAL  }
                    OPTIONAL,
    crlExtensions             [0]  Extensions OPTIONAL  }

      Version  ::= INTEGER  {  v1(0), v2(1)  }
```

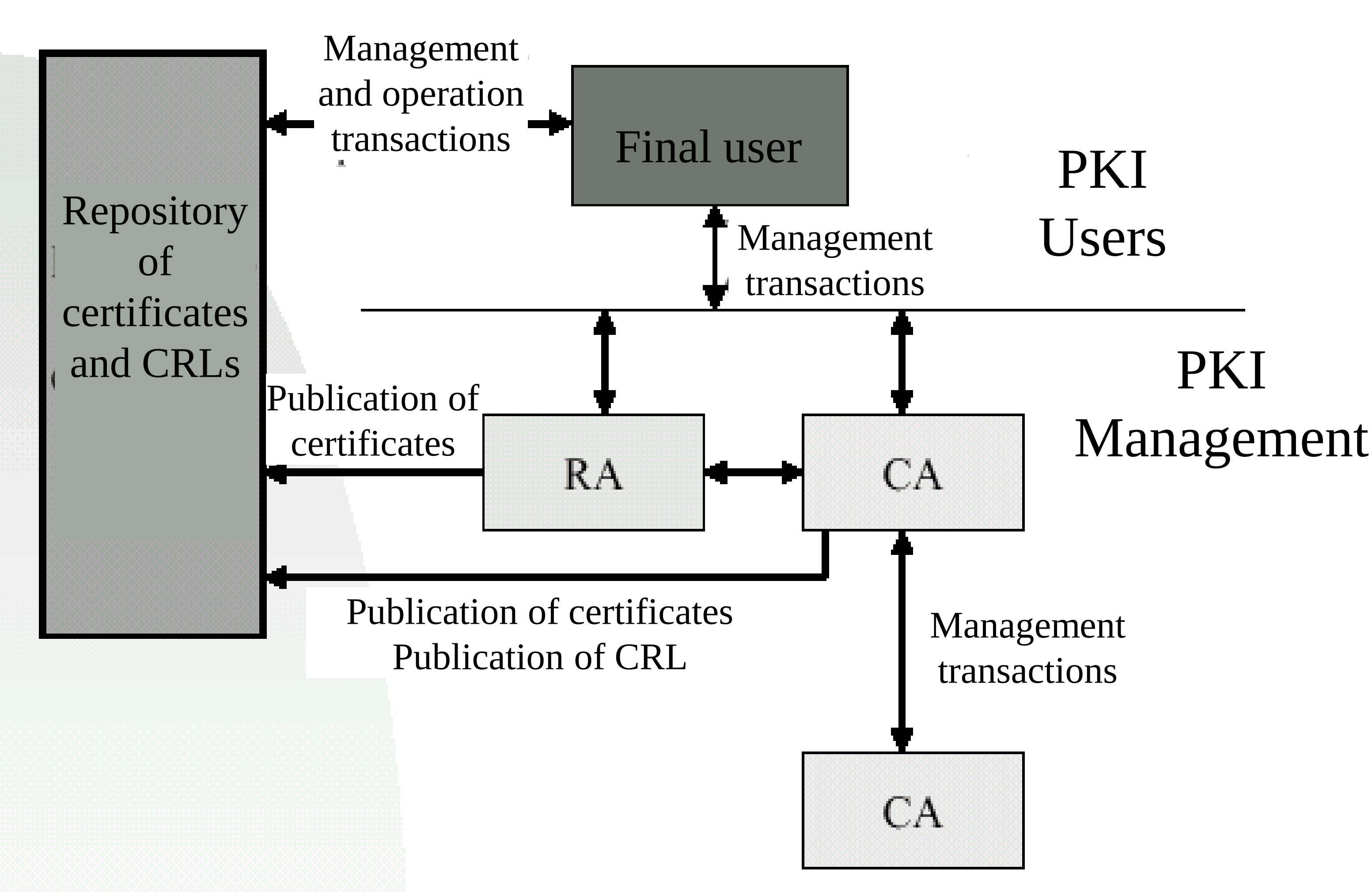
Types of X.509 certificates

- **Up to now we have explained the certificates of names (PKC):**
  - My identity (and public key) signed by the CA
  - Useful for authentication
- **There is a second kind of certificates: attribute certificates (AC).**
  - Associates attributes to a PKC.
  - Useful for authorization
  - Issued by a attribute authority (AA)

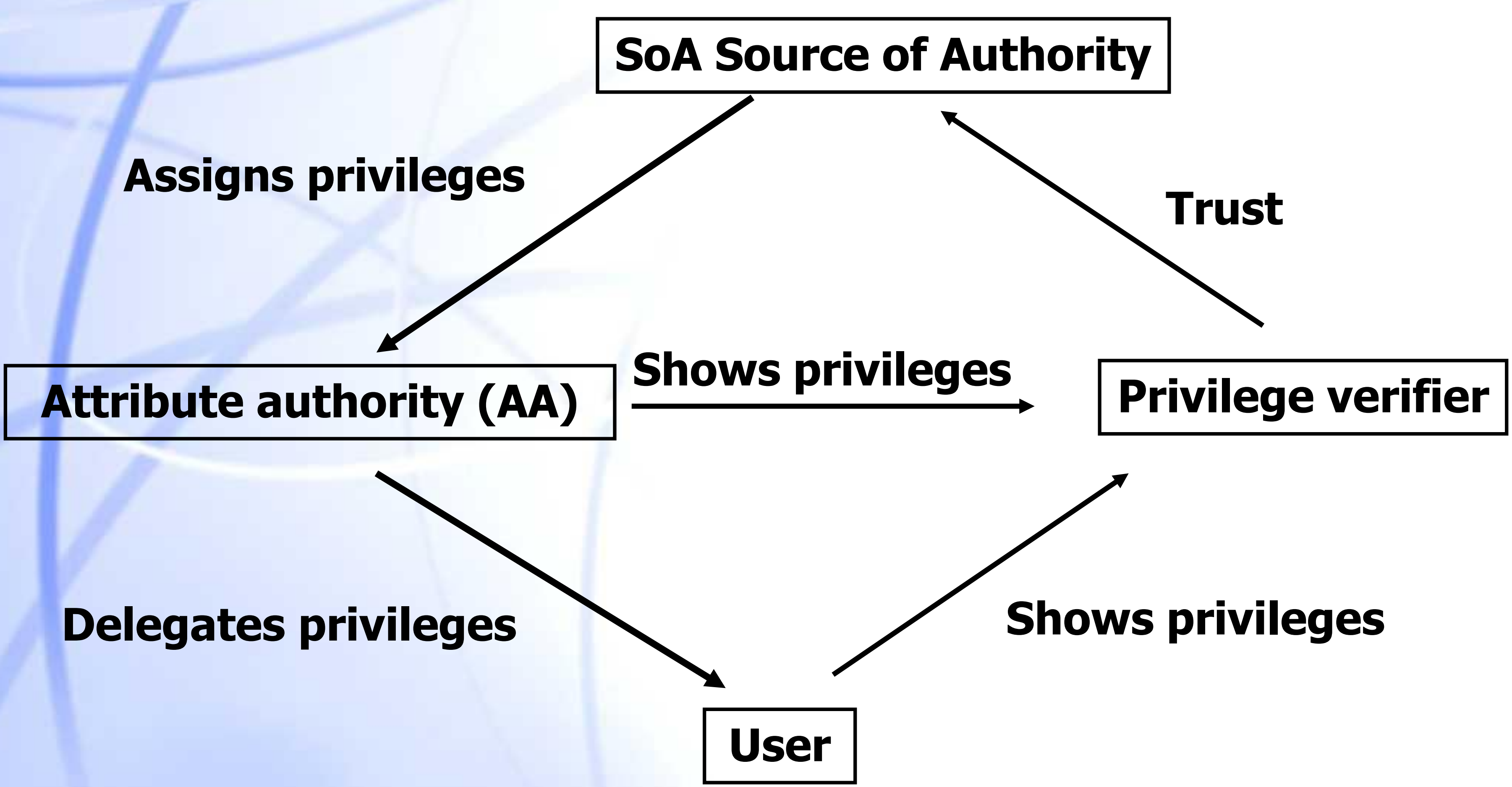
Two types of certificates?

- **Usually, the identity of an entity is:**
  - Permanent in time
  - Independent of the place
- **The authorization by an entity to use resources:**
  - It is locally issued and depends of the location
  - Changes with time

PKI (Public Key Infrastructure)


Repository of certificates and CRLs
Management and operation transactions
Final user
Management transactions
PKI Users
PKI Management
Publication of certificates
RA
CA
Publication of certificates
Publication of CRL
Management transactions
CA

PMI (Privilege Management Infrastructure)



SoA Source of Authority

Assigns privileges

Trust

Attribute authority (AA)

Shows privileges

Privilege verifier

Delegates privileges

Shows privileges

User

Attribute certificates

```
AttributeCertificate ::= SEQUENCE {
  toBeSigned            AttributeCertificateInfo,
  algorithmIdentifier   AlgorithmIdentifier,
  encrypted              BIT STRING
}

AttributeCertificateInfo ::= SEQUENCE {
  version               AttCertVersion, -- version is v2
  holder                Holder,
  issuer                AttCertIssuer,
  signature             AlgorithmIdentifier,
  serialNumber          CertificateSerialNumber,
  attrCertValidityPeriod  AttCertValidityPeriod,
  attributes            SEQUENCE OF Attribute,
  issuerUniqueID        UniqueIdentifier OPTIONAL,
  extensions            Extensions OPTIONAL
}
```

References to the name certificate

Standard attributes

- **Some attributes are defined as standard:**
  - Authentication information for the service
    - ✓ Credentials to use the service
  - Access identity
    - ✓ Used by the verifier of attribute certificates
  - Group and role
    - ✓ Define the membership of the owner of the certificate to a group

Privilege delegation

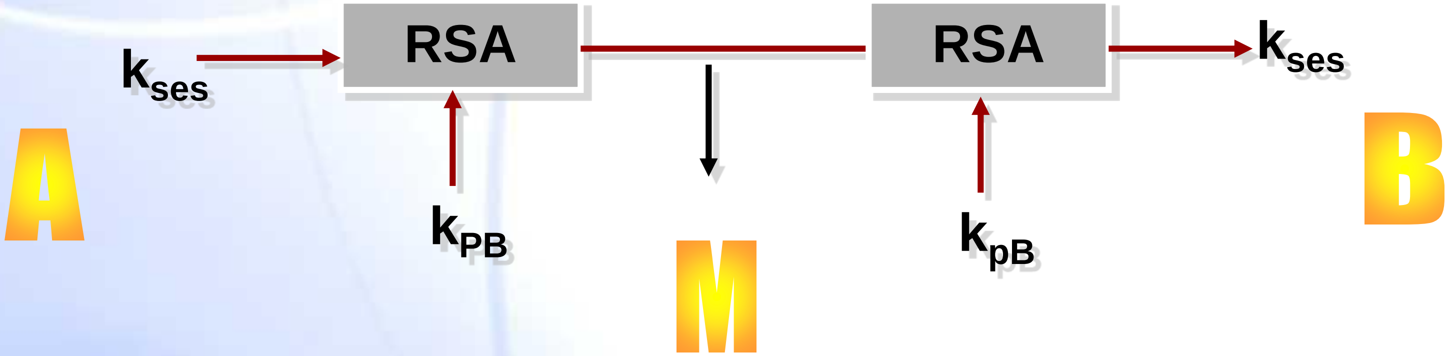
- **Similarly to PKC a chain of attribute certificate can be established**
- **Using extensions in the attribute certificates, the SoA or an AA (with privileges) can sign an attribute certificate which allows the receiver to become, in turn, an AA.**
  - You can limit the delegation depth

Exchange of Session Keys using Asymmetric Algorithms

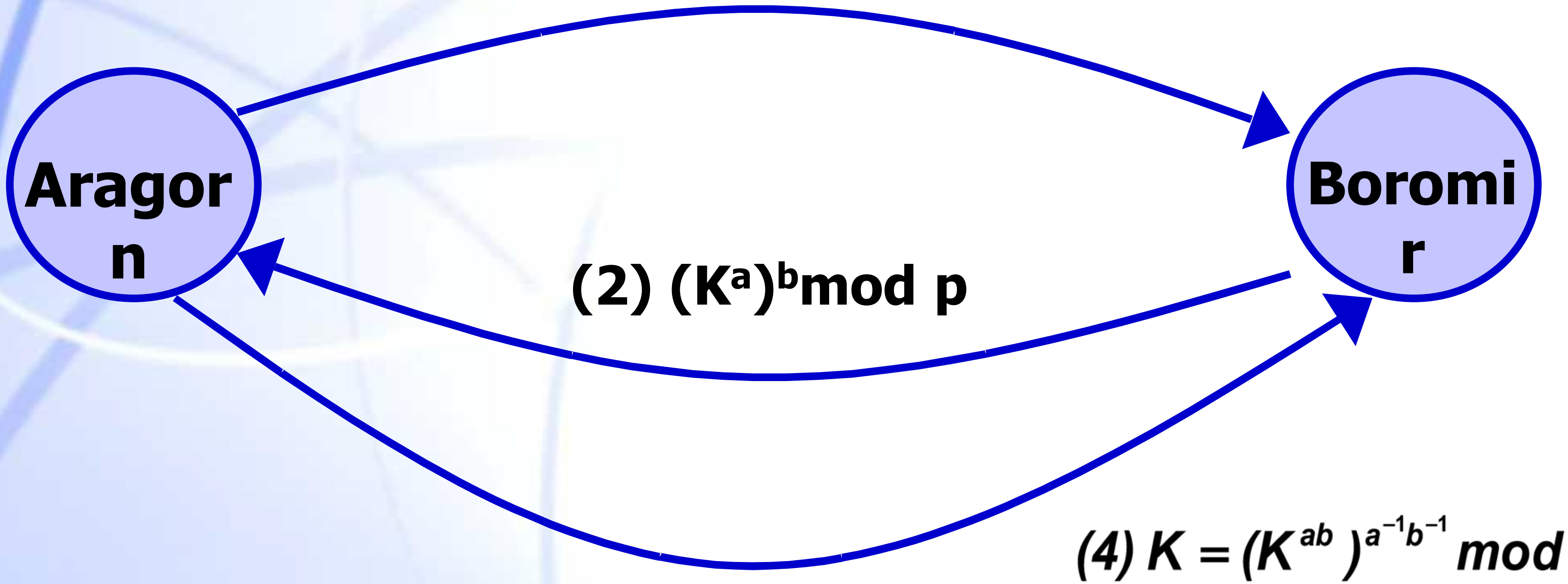
Departamento de INGENIERIA TELEMATICA
www.it.uc3m.es

Encryption with Asymmetric Keys

- Can use any of the encryption methods from chapter 4 to encrypt session keys to be sent to the receiver
- For example:



Shamir



$1 \leq a , b \leq p-2$
coprimes (p-1)

(1) $(K^a)\text{mod } p$

Aragorn

Boromir

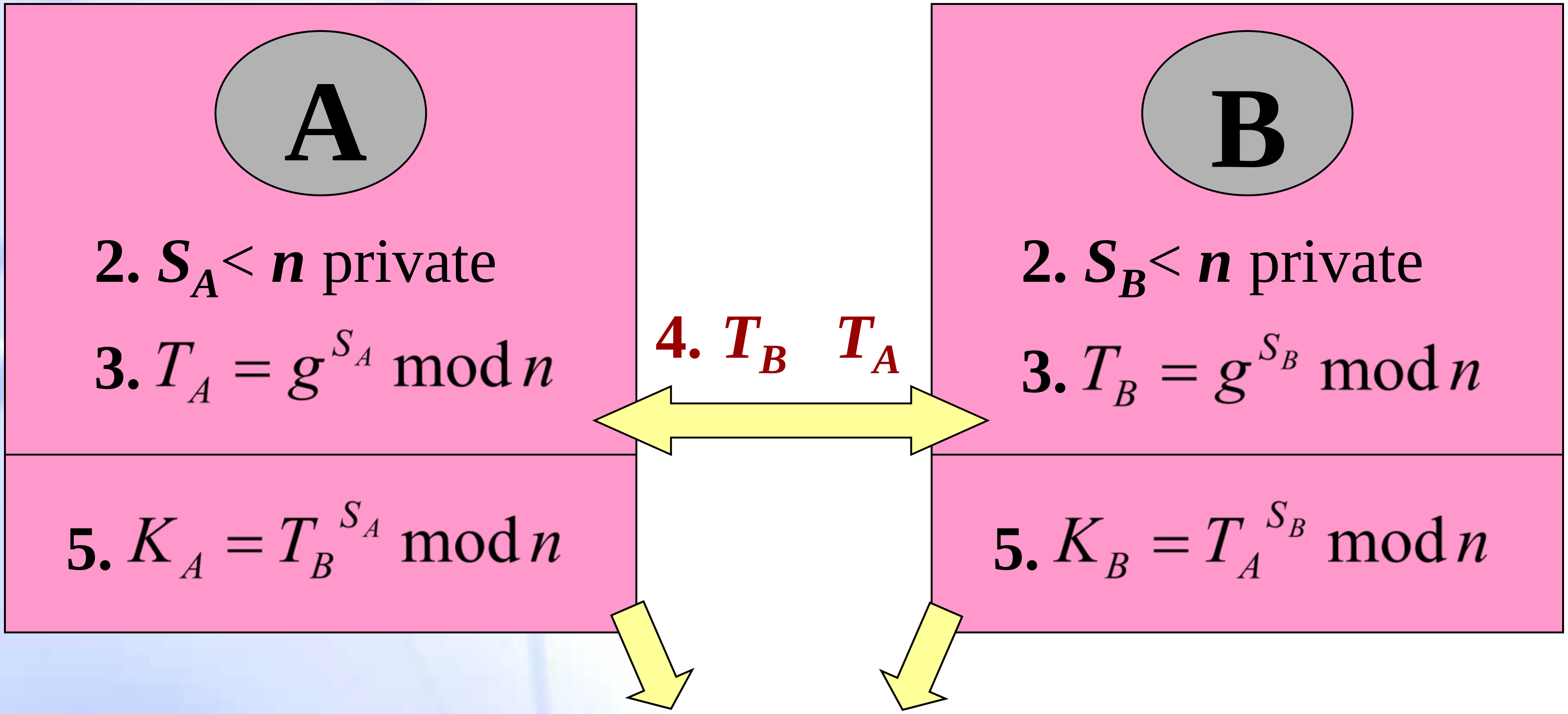
(2) $(K^a)^b\text{mod } p$

(3) $(K^{ab})^{a^{-1}}\text{mod } p$

$$(4)\ K = (K^{ab})^{a^{-1}b^{-1}} \ mod\ p$$

Diffie-Hellman Key Exchange

1. **Selection of a prime number `n` (public) and `g` a primitive root of `n`**



A

2. $S_A < n$ private

3. $T_A = g^{S_A} \bmod n$

5. $K_A = T_B^{S_A} \bmod n$

4. T_B T_A

B

2. $S_B < n$ private

3. $T_B = g^{S_B} \bmod n$

5. $K_B = T_A^{S_B} \bmod n$

$$K_A = T_B^{S_A} = (g^{S_B})^{S_A} = (g^{S_A})^{S_B} = T_A^{S_B} = K_B$$