Chapter3

Public-Key Cryptography and Message Authentication

OUTLINE

- Approaches to Message Authentication
- Secure Hash Functions and HMAC
- Public-Key Cryptography Principles
- Public-Key Cryptography Algorithms
- Digital Signatures
- Key Management

Authentication

- Requirements - must be able to verify that:
  1. Message came from apparent source or author,
  2. Contents have not been altered,
  3. Sometimes, it was sent at a certain time or sequence.

- Protection against active attack (falsification of data and transactions)

Approaches to Message Authentication

- Authentication Using Conventional Encryption
  - Only the sender and receiver should share a key
- Message Authentication without Message Encryption
  - An authentication tag is generated and appended to each message
- Message Authentication Code
  - Calculate the MAC as a function of the message and the key. MAC = F(K, M)

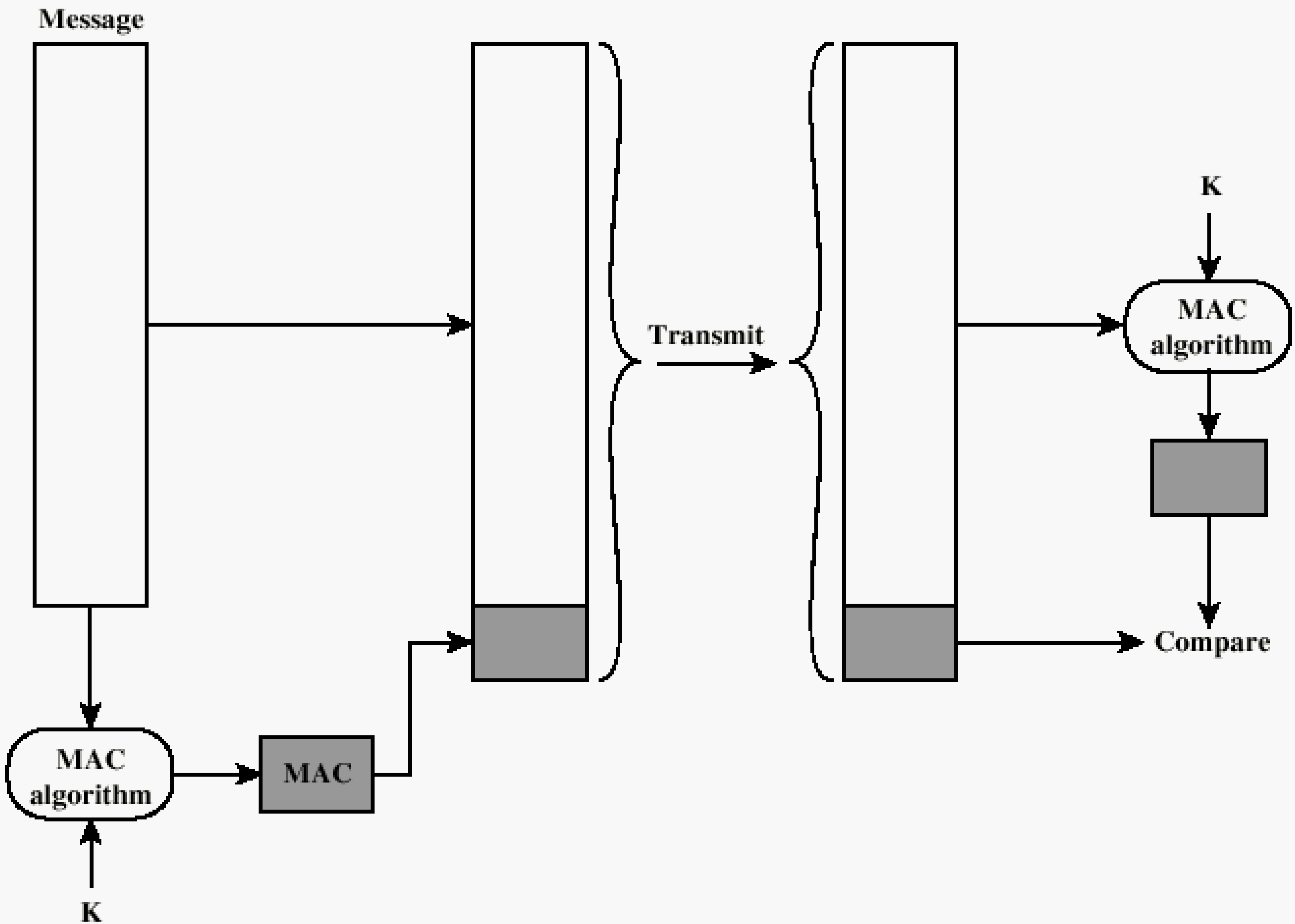
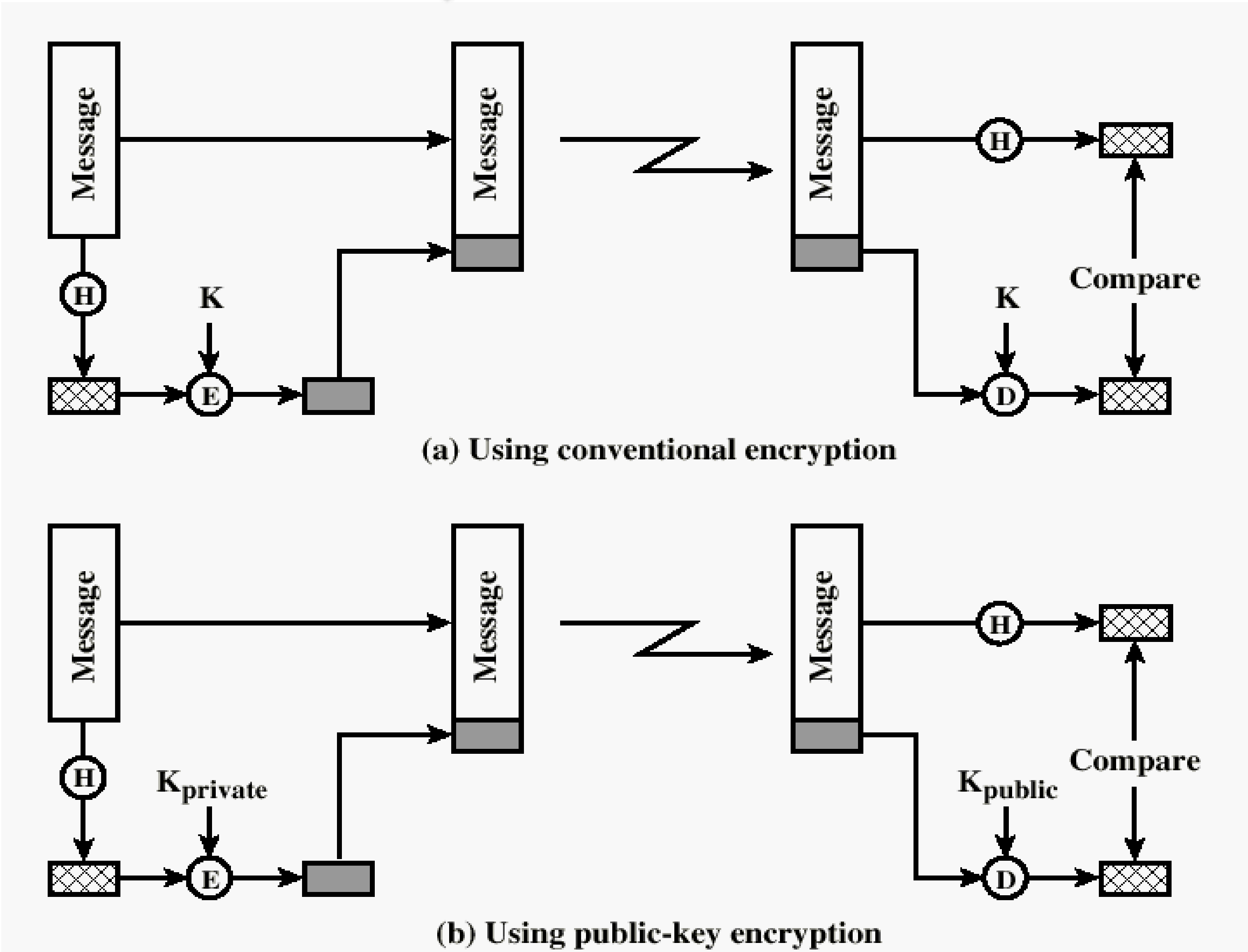


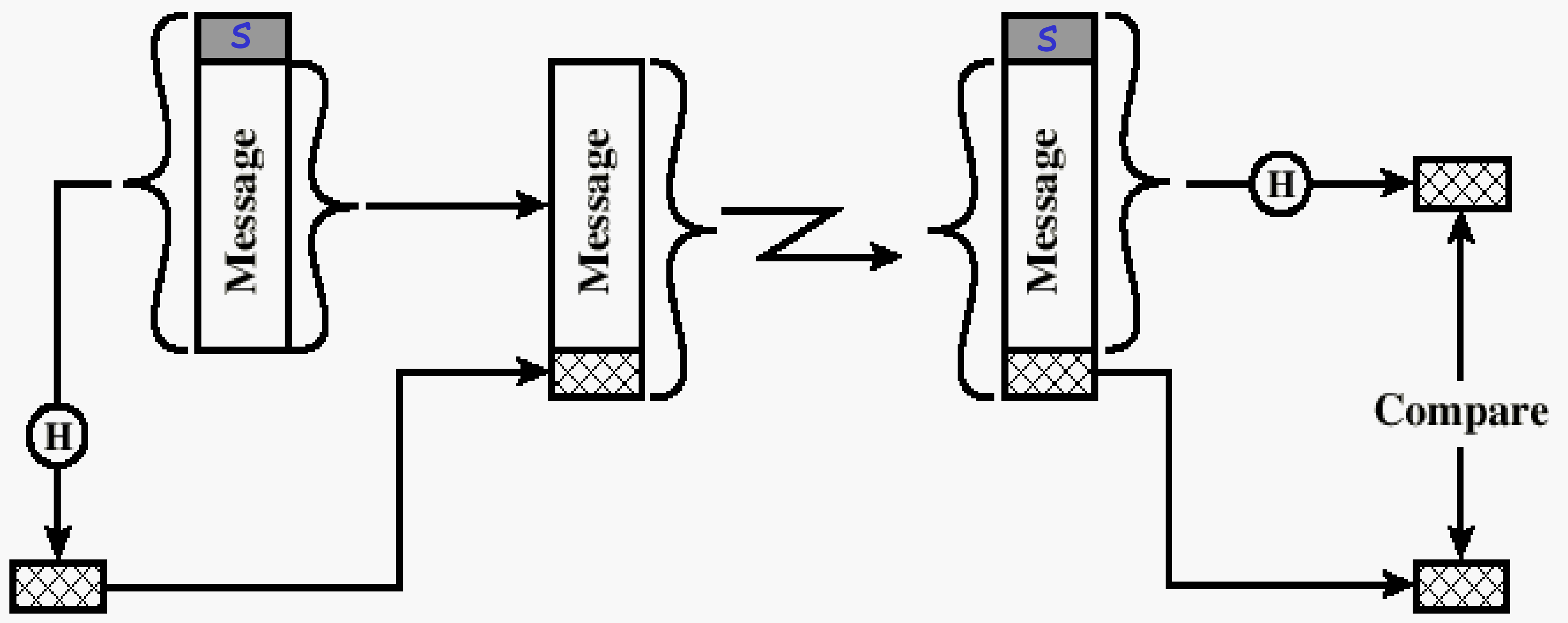
Figure 3.1 Message Authentication Using a Message Authentication Code (MAC)

One-way HASH function



One-way HASH function

- Secret value is added before the hash and removed before transmission.
  - Variation of this technique is HMAC



(c) Using secret value

Secure HASH Functions

- Purpose of the HASH function is to produce a ”fingerprint.
- Properties of a HASH function H :

1. H can be applied to a block of data at any size
2. H produces a fixed length output
3. H(x) is easy to compute for any given x.
4. For any given block x, it is computationally infeasible to find x such that H(x) = h
5. For any given block x, it is computationally infeasible to find $y \neq x$ with H(y) = H(x).
6. It is computationally infeasible to find any pair (x, y) such that H(x) = H(y)

Simple Hash Function

	bit 1	bit 2	• • •	bit *n*
block 1	b_{11}	b_{21}		b_{n1}
block 2	b_{12}	b_{22}		b_{n2}
	• • •	• • •	• • •	• • •
block *m*	b_{1m}	b_{2m}		b_{nm}
hash code	C_1	C_2		C_n

Figure 3.3 Simple Hash Function Using Bitwise XOR

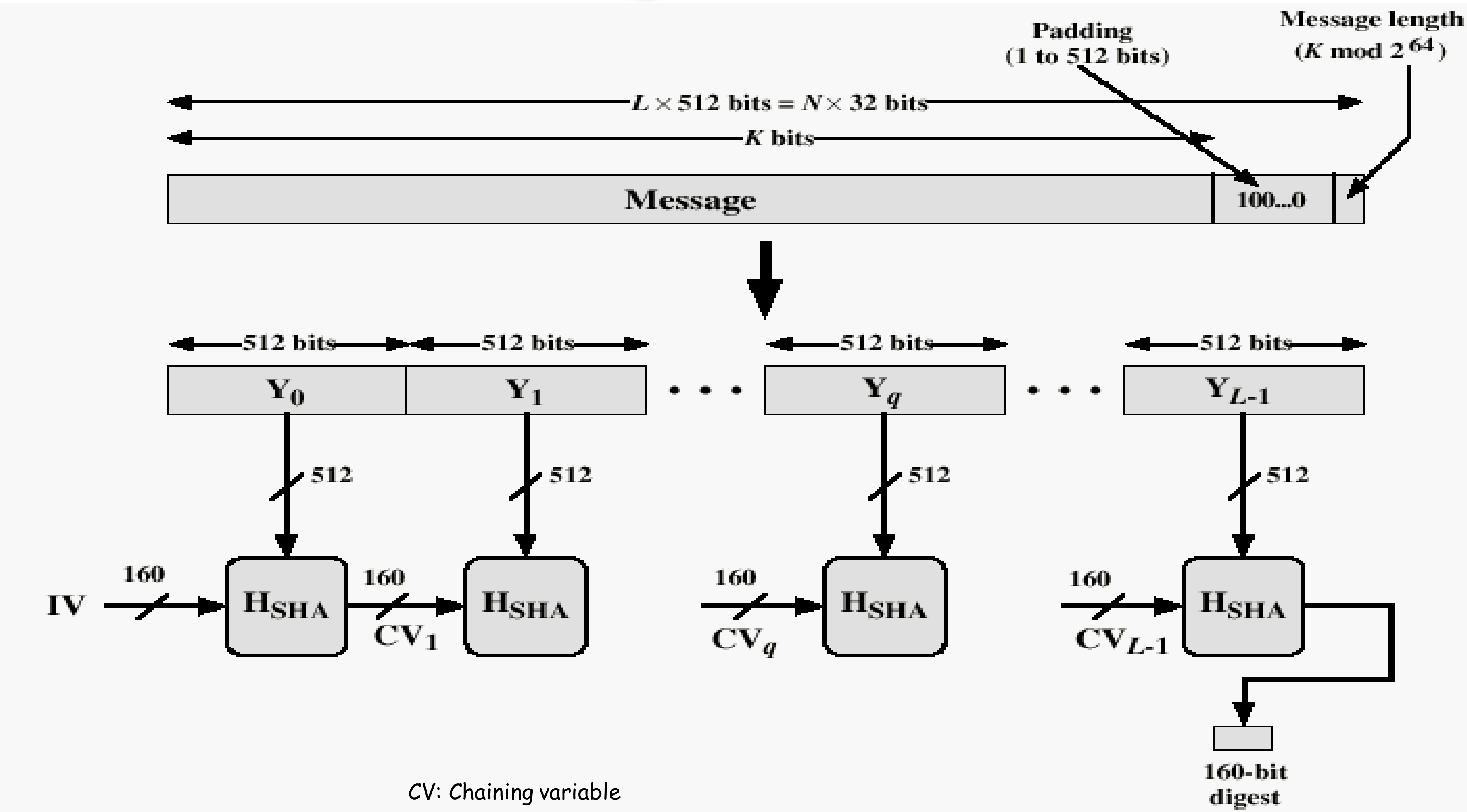
- C_i=b_{i1} XOR b_{i2} XOR ...XOR b_{im}

Where,

C_i=hash value

b_{ij}=ith bit of j block

Message Digest Generation Using SHA-1



SHA-1 Processing of single 512-Bit Block

A= 67452301
B= EFCDAB89
C= 98BADCFE
D= 10325476
E= C3D2E1F0

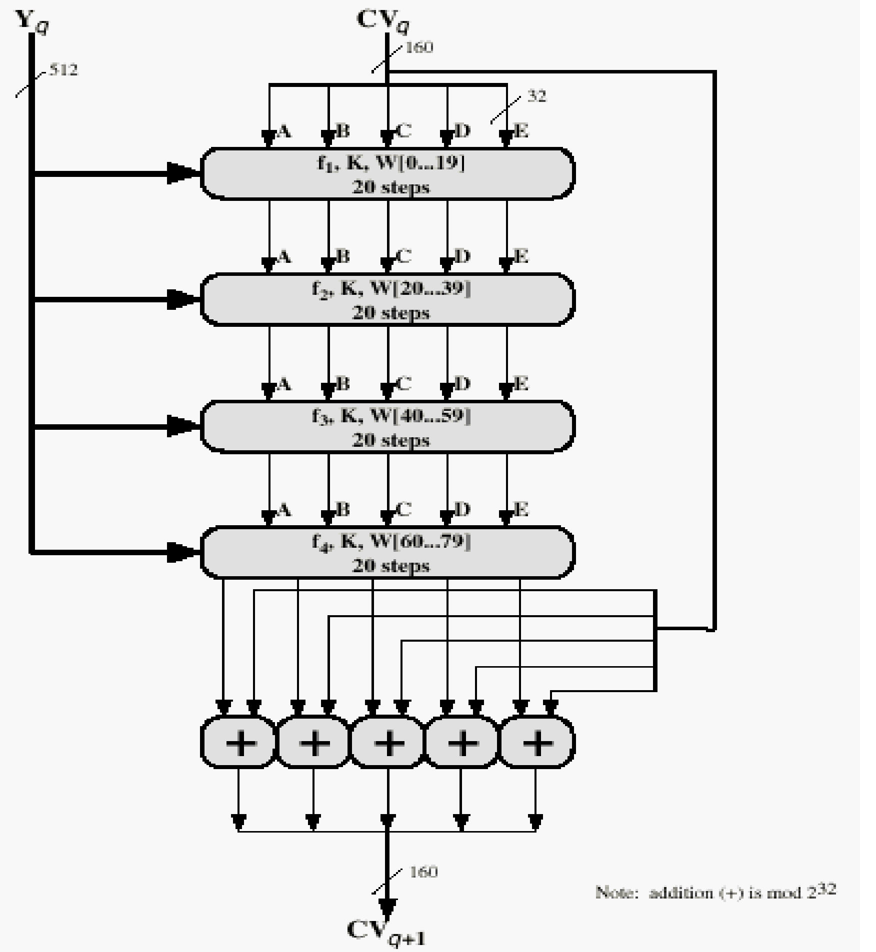


Figure 3.5 SHA-1 Processing of a Single 512-bit Block

Comparison of SHA Parameters

Table 12.1 Comparison of SHA Parameters

	SHA-1	SHA-256	SHA-384	SHA-512
Message digest size	160	256	384	512
Message size	$< 2^{64}$	$< 2^{64}$	$< 2^{128}$	$< 2^{128}$
Block size	512	512	1024	1024
Word size	32	32	64	64
Number of steps	80	64	80	80
Security	80	128	192	256

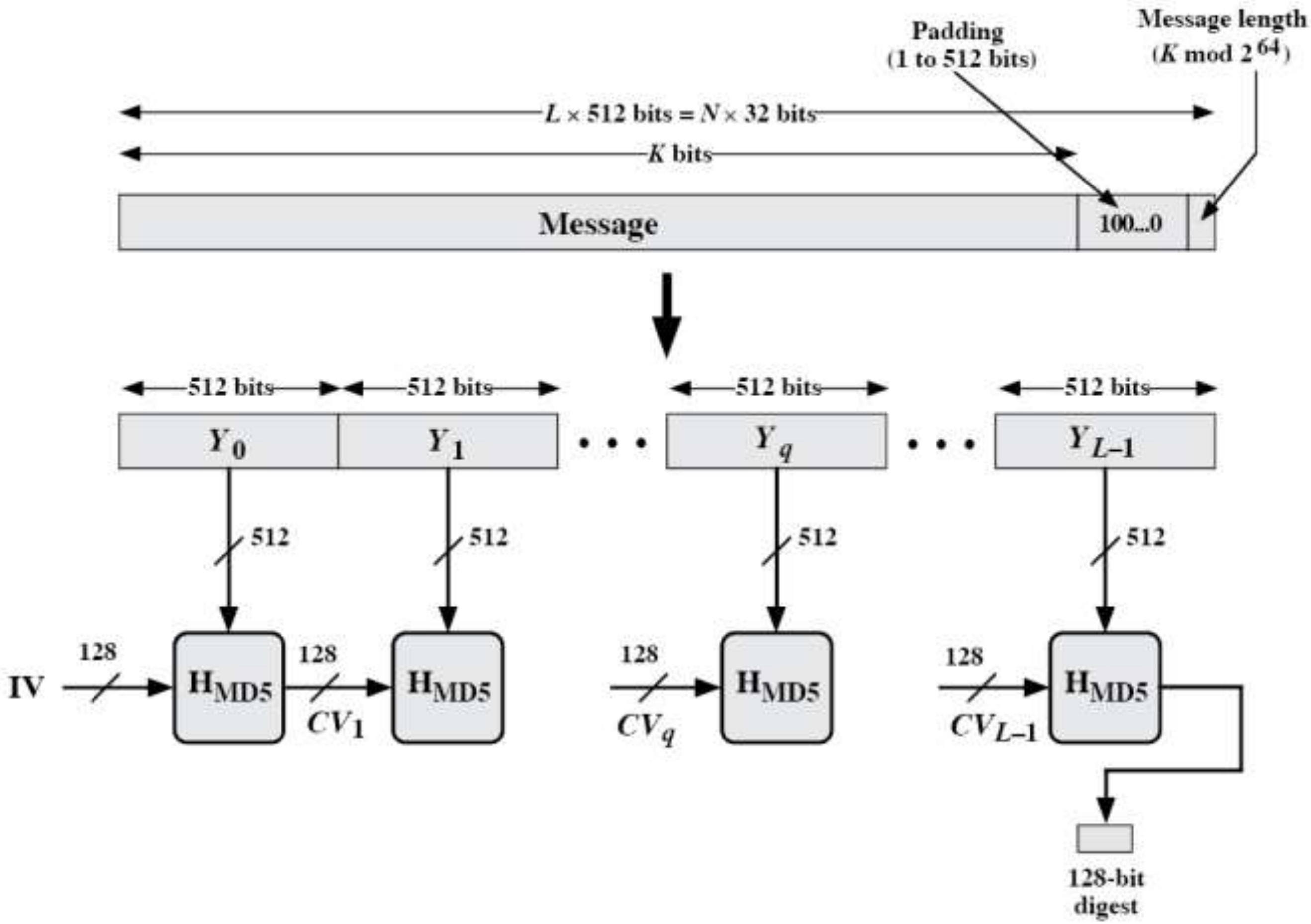
MD5

- Developed by Ron Rivest at MIT
- Input: a message of arbitrary length
- Output: 128-bit message digest
- 32-bit word units, 512-bit blocks
- Son of MD2, MD4

MD5

- MD5 processes a variable length message into a fixed-length output of 128 bits.
- The input message is broken up into chunks of 512-bit blocks; the message is **padded** so that its length is divisible by 512.
- The remaining bits are filled up with a 64-bit integer representing the length of the original message.
- The main MD5 algorithm operates on a 128-bit state, divided into four 32-bit words, denoted *A*, *B*, *C* and *D*.
- These are initialized to certain fixed constants.

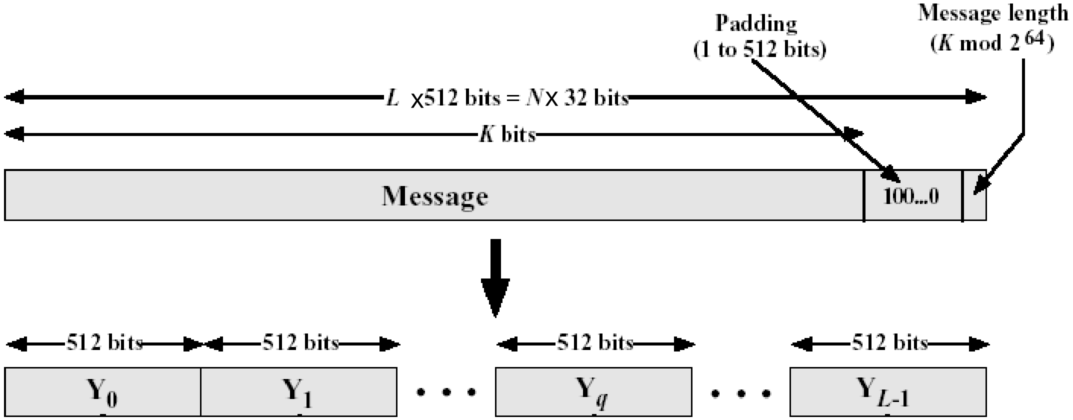
MD5



Message Digest Generation Using MD5

MD5 Logic

- Step 1: Append padding bits
  - Padded so that its bit length ≡ 448 mod 512 (i.e., the length of padded message is 64 bits less than an integer multiple of 512 bits)
  - Padding is always added, even if the message is already of the desired length (1 to 512 bits)
  - Padding bits: 1000….0 (a single 1-bit followed by the necessary number of 0-bits)
- Step 2: Append length
  - 64-bit length: contains the length of the original message modulo 2^{64}



  - The expanded message is $Y_0, Y_1, \ldots, Y_{L-1}$; the total length is $L \times 512$ bits
  - The expanded message can be thought of as a multiple of 16 32-bit words
  - Let M[0 … N-1] denote the word of the resulting message, where $N = L \times 16$

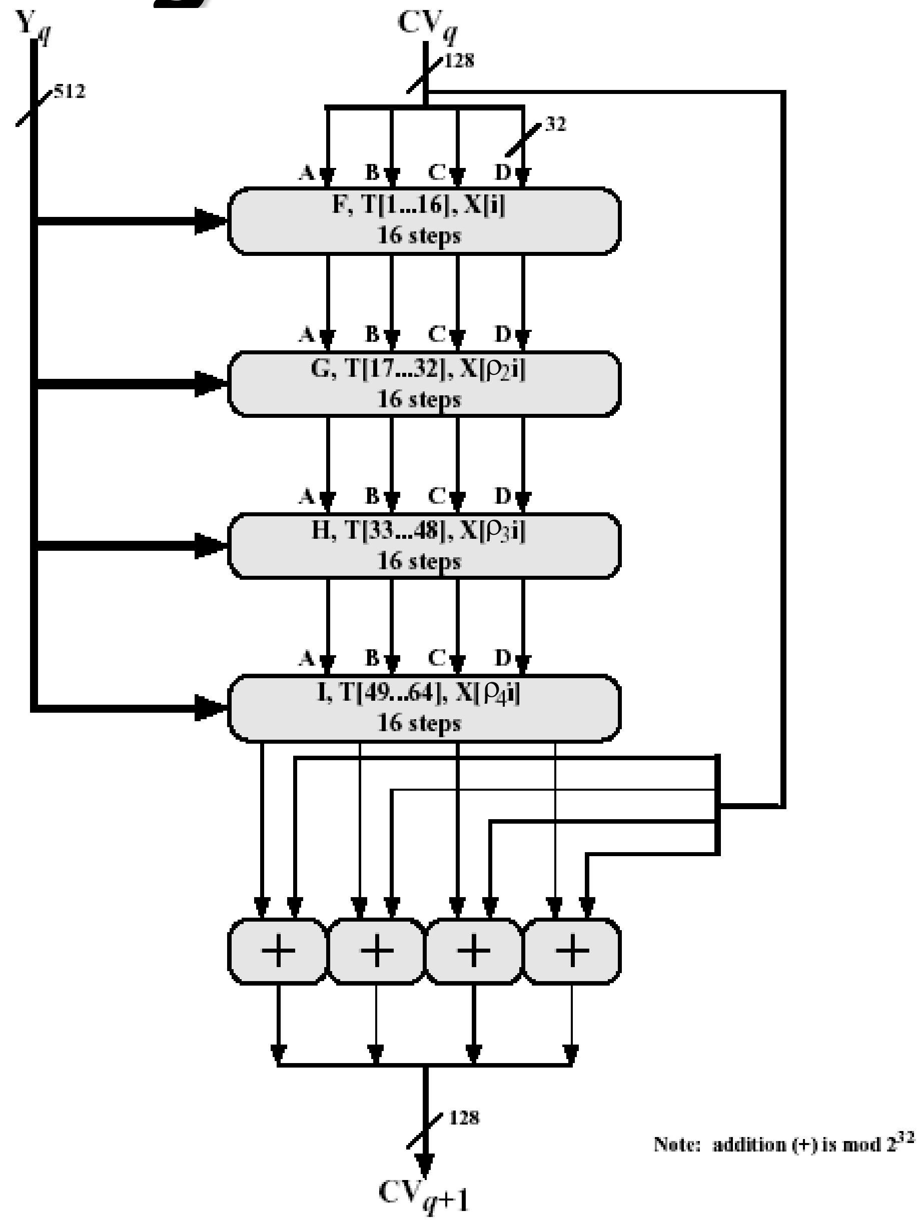
MD5 Logic

- Step 3: Initialize MD buffer
  - 128-bit buffer (four 32-bit registers A,B,C,D) is used to hold intermediate and final results of the hash function
  - A,B,C,D are initialized to the following values
    - A = 67452301, B = EFCDAB89, C = 98BADCFE, D = 10325476
    - Stored in *little-endian* format (least significant byte of a word in the low-address byte position)
      - E.g. word A: 01 23 45 67 (low address … high address)
- Step 4: Process message in 512-bit (16-word) blocks
  - Heart of the algorithm called a *compression function*
  - Consists of 4 rounds
  - The 4 rounds have a similar structure, but each uses a different *primitive logical functions*, referred to as F, G, H, and I
  - Each round takes as input the current 512-bit block (Y_q), 128-bit buffer value ABCD and updates the contents of the buffer
  - Each round also uses the table T[1 … 64], constructed from the sine function; $T[i] = 2^{32} \times abs(sin(i))$
  - The output of 4^{th} round is added to the CV_q to produce CV_{q+1}

MD5 Logic

MD5 processing of a single 512-bit block

(MD5 compression function)



MD5 Logic

- **Table T, constructed from the sine function**
  - **T[i] = integer part of $2^{32} \times$ abs(sin(i)), where i is in radians**

T[1] = D76AA478	T[17] = F61E2562	T[33] = FFFA3942	T[49] = F4292244
T[2] = E8C7B756	T[18] = C040B340	T[34] = 8771F681	T[50] = 432AFF97
T[3] = 242070DB	T[19] = 265E5A51	T[35] = 699D6122	T[51] = AB9423A7
T[4] = C1BDCEEE	T[20] = E9B6C7AA	T[36] = FDE5380C	T[52] = FC93A039
T[5] = F57C0FAF	T[21] = D62F105D	T[37] = A4BEEA44	T[53] = 655B59C3
T[6] = 4787C62A	T[22] = 02441453	T[38] = 4BDECFA9	T[54] = 8F0CCC92
T[7] = A8304613	T[23] = D8A1E681	T[39] = F6BB4B60	T[55] = FFEFF47D
T[8] = FD469501	T[24] = E7D3FBC8	T[40] = BEBFBC70	T[56] = 85845DD1
T[9] = 698098D8	T[25] = 21E1CDE6	T[41] = 289B7EC6	T[57] = 6FA87E4F
T[10] = 8B44F7AF	T[26] = C33707D6	T[42] = EAA127FA	T[58] = FE2CE6E0
T[11] = FFFF5BB1	T[27] = F4D50D87	T[43] = D4EF3085	T[59] = A3014314
T[12] = 895CD7BE	T[28] = 455A14ED	T[44] = 04881D05	T[60] = 4E0811A1
T[13] = 6B901122	T[29] = A9E3E905	T[45] = D9D4D039	T[61] = F7537E82
T[14] = FD987193	T[30] = FCEFA3F8	T[46] = E6DB99E5	T[62] = BD3AF235
T[15] = A679438E	T[31] = 676F02D9	T[47] = 1FA27CF8	T[63] = 2AD7D2BB
T[16] = 49B40821	T[32] = 8D2A4C8A	T[48] = C4AC5665	T[64] = EB86D391

MD5 Logic

- Step 5: Output
  - After all L 512-bit blocks have been processed, the output from the L^th^ stage is the 128-bit message digest
  - $CV_0 = IV$

    $CV_{q+1} = SUM_{32}(CV_q, RF_I[Y_q, RF_H[Y_q, RF_G[Y_q, RF_F[Y_q, CV_q]]]])$

    $MD = CV_L$

    where

    IV = initial value of the ABCD buffer, defined in step 3

    Y_q = the q^th^ 512-bit block of the message

    L = the number of blocks in the message (including padding and length fields)

    CV_q = chaining variable processed with the q^th^ block of the message

    RF_x = round function using primitive logical function x

    MD = final message digest value

    SUM_{32} = addition modulo 2^{32} performed separately on each word

MD5 Compression Function

- Each round consists of a sequence of 16 steps operating on the buffer ABCD
- Each step is of the form

  $b \leftarrow b + ((a + g(b, c, d) + X[k] + T[i] <<< s)$

  Where

  a,b,c,d = the 4 words of the buffer, in a specified order that varies across steps

  g = one of the primitive functions F, G, H, I

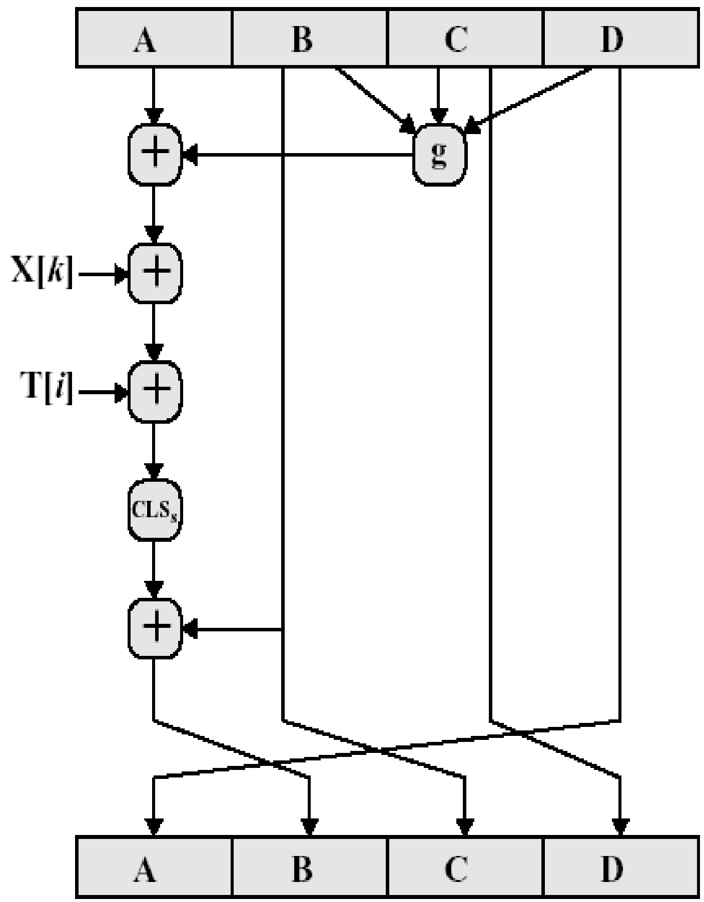
  <<< s = circular left shift (rotation) of the 32-bit arguments by s bits

  $X[k] = M[q \times 16 + k]$ = the k^{th} 32-bit word in the q^{th} 512-bit block of the message

  T[i] = the i^{th} 32-bit word in table T

  + = addition modulo 2^{32}

Elementary MD5 Operation (Single Step)



MD5 Primitive Logical Functions

- One of the 4 primitive logical functions is used in each 4 rounds of the algorithm
- Each primitive function takes three 32-bit words as input and produces a 32-bit word output
- Each function performs a set of bitwise logical operations

Round	Primitive function g	g(b, c, d)
1	F(b, c, d)	$(b \wedge c) \vee (b' \wedge d)$
2	G(b, c, d)	$(b \wedge d) \vee (c \wedge d')$
3	H(b, c, d)	$b \oplus c \oplus d$
4	I(b, c, d)	$c \oplus (b \vee d')$

Truth table

b	c	d	F	G	H	I
0	0	0	0	0	0	1
0	0	1	1	0	1	0
0	1	0	0	1	1	0
0	1	1	1	0	0	1
1	0	0	0	0	1	1
1	0	1	0	1	0	1
1	1	0	1	1	0	0
1	1	1	1	1	1	0

X[k]

- The array of 32-bit words X[0..15] holds the value of current 512-bit input block being processed
- Within a round, each of the 16 words of X[i] is used exactly once, during one step
  - The order in which these words is used varies from round to round
  - In the first round, the words are used in their original order
  - For rounds 2 through 4, the following permutations are used
    - $\rho_2(i) = (1 + 5i) \bmod 16$
    - $\rho_3(i) = (5 + 3i) \bmod 16$
    - $\rho_4(i) = 7i \bmod 16$

MD5

- var int[64] r, k
- r[0..15] := {7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22}
- r[16..31] := {5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20}
- r[32..47] := {4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23}
- r[48..63] := {6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21}

- //Use binary integer part of the sines of integers as constants:
- for i from 0 to 63
-     k[i] := floor(abs(sin(i + 1)) × 2^32)

- //Initialize variables:
- var int h0 := 0x67452301
- var int h1 := 0xEFCDAB89
- var int h2 := 0x98BADCFE
- var int h3 := 0x10325476

- //Pre-processing:
- append "1" bit to message
- append "0" bits until message length in bits ≡ 448 (mod 512)
- append bit length of message as 64-bit little-endian integer to message

* Refer to RFC 1321

MD5

- //Process the message in successive 512-bit chunks:
- for each 512-bit chunk of message
-     break chunk into sixteen 32-bit little-endian words w(i), 0 ≤ i ≤ 15

-     //Initialize hash value for this chunk:
-     var int a := h0
-     var int b := h1
-     var int c := h2
-     var int d := h3

-     //Main loop:
-     for i from 0 to 63
-         if 0 ≤ i ≤ 15 then
-             f := (b and c) or ((not b) and d)
-             g := i
-         else if 16 ≤ i ≤ 31
-             f := (d and b) or ((not d) and c)
-             g := (5×i + 1) mod 16
-         else if 32 ≤ i ≤ 47
-             f := b xor c xor d
-             g := (3×i + 5) mod 16
-         else if 48 ≤ i ≤ 63
-             f := c xor (b or (not d))
-             g := (7×i) mod 16

MD5

- temp := d
- d := c
- c := b
- b := ((a + f + k(i) + w(g)) leftrotate r(i)) + b
- a := temp
- //end of main loop
- //Add this chunk's hash to result so far:
- h0 := h0 + a
- h1 := h1 + b
- h2 := h2 + c
- h3 := h3 + d

- var int digest := h0 append h1 append h2 append h3 //(expressed as little-endian)

Other Secure HASH functions

	SHA-1	MD5	RIPEMD-160
Digest length	160 bits	128 bits	160 bits
Basic unit of processing	512 bits	512 bits	512 bits
Number of steps	80 (4 rounds of 20)	64 (4 rounds of 16)	160 (5 paired rounds of 16)
Maximum message size	2^{64}-1 bits	∞	∞

HMAC

- Use a MAC derived from a cryptographic hash code, such as SHA-1.
- **Motivations:**
  - Cryptographic hash functions executes faster in software than encryptoin algorithms such as DES
  - Library code for cryptographic hash functions is widely available

HMAC

- specified as Internet standard RFC2104
  - The mandatory-to-implement MAC for IP security
- uses hash function on the message:

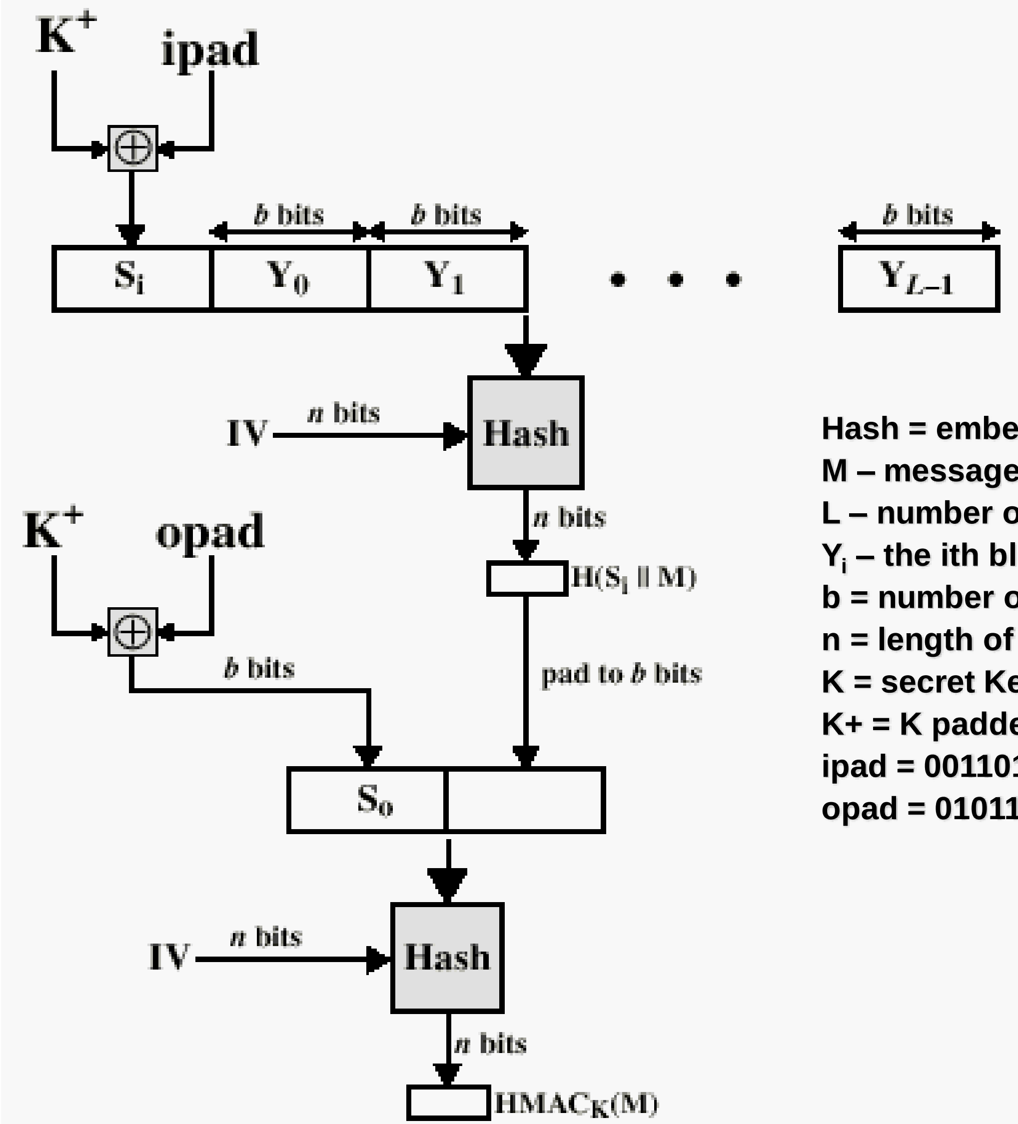
  $HMAC_K$ = Hash[(K^+ XOR opad) ||
  Hash[(K^+ XOR ipad)||M)]]
- where K^+ is the key padded out to size
- and opad, ipad are specified padding constants

  -ipad =00110110 (36 in hex) repeated b/8 times

  -opad=01011100 (5C in hex) repeated b/8 times

  (b is number of bits in a block)
- any hash function can be used
  - E.g., MD5, SHA-1, RIPEMD-160, Whirlpool

HMAC Structure

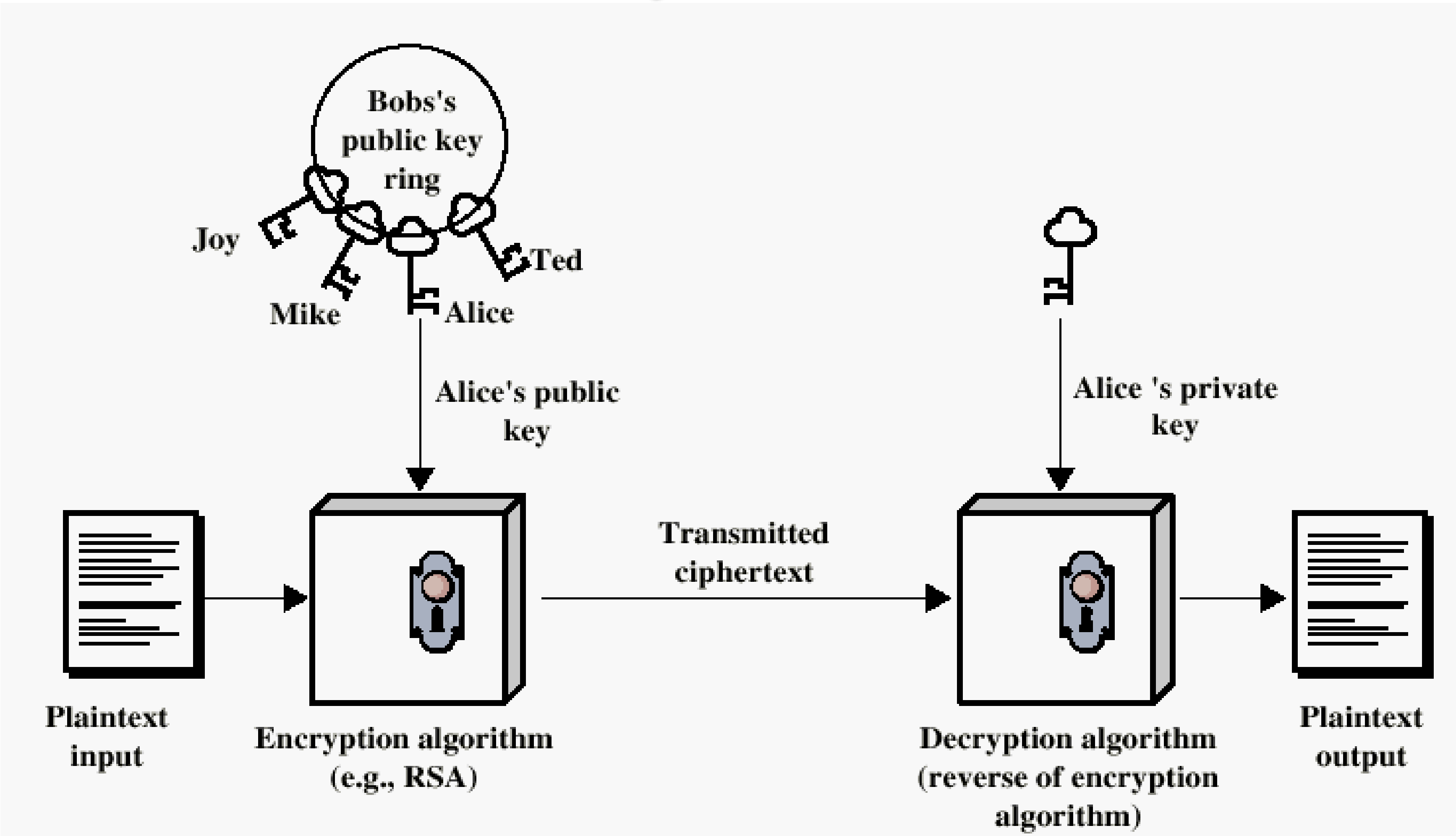


Hash = embedded hash function (e.g., SHA-1)
M – message
L – number of blocks in M
Y_i – the ith block of M   0 < i < L
b = number of bits in a block
n = length of hash code produced by embedded hash
K = secret Key
K+ = K padded on left with zeros so length is b
ipad = 00110110 repeated b/8 times
opad = 01011100 repeated b/8 times

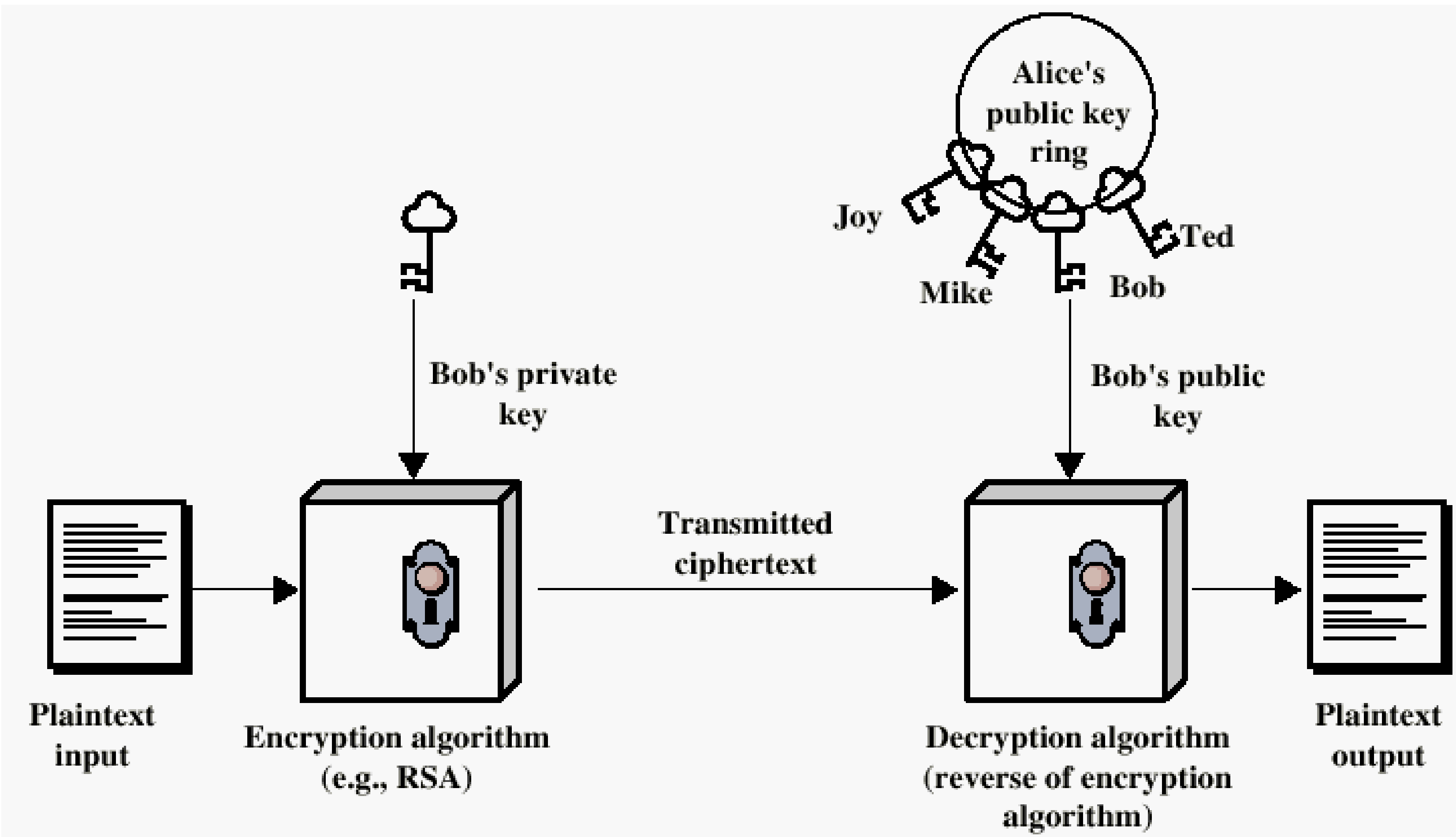
Public-Key Cryptography Principles

- The use of two keys has consequences in: key distribution, confidentiality and authentication.
- The scheme has six ingredients
  - Plaintext
  - Encryption algorithm
  - Public and private key
  - Ciphertext
  - Decryption algorithm

Encryption using Public-Key system



Authentication using Public-Key System



Applications for Public-Key Cryptosystems

- Three categories:
  - **Encryption/decryption:** The sender encrypts a message with the recipient’s public key.
  - **Digital signature:** The sender ”signs” a message with its private key.
  - **Key echange:** Two sides cooperate to exchange a session key.

Requirements for Public-Key Cryptography

1. Computationally easy for a party B to generate a pair (public key KU_b, private key KR_b)
2. Easy for sender to generate ciphertext: $C = E_{KUb}(M)$
3. Easy for the receiver to decrypt ciphertext using private key:

$$M = D_{KRb}(C) = D_{KRb}[E_{KUb}(M)]$$

Requirements for Public-Key Cryptography

4. Computationally infeasible to determine private key (KR_b) knowing public key (KU_b)
5. Computationally infeasible to recover message M, knowing KU_b and ciphertext C
6. Either of the two keys can be used for encryption, with the other used for decryption:

$$M = D_{KRb}[E_{KUb}(M)] = D_{KUb}[E_{KRb}(M)]$$

Public-Key Cryptographic Algorithms

- RSA and Diffie-Hellman - Stanford
- **RSA** - Ron Rivest, Adi Shamir and Len Adleman at MIT, in 1977.
  - RSA is a block cipher
  - The most widely implemented
- **Diffie-Hellman**
  - Echange a secret key securely
  - Compute discrete logarithms

Private-Key Cryptography

- traditional **private/secret/single-key** cryptography uses **one** key
- shared by both sender and receiver
- if this key is disclosed communications are compromised
- also is **symmetric**, parties are equal
- hence does not protect sender from receiver forging a message & claiming the message sent by sender

Public-Key Cryptography

- uses **two** keys – a public & a private key
- **asymmetric** since parties are **not** equal
- uses clever application of number theoretic concepts to function
- complements **rather than** replaces private key crypto

Public-Key Cryptography

- **public-key/two-key/asymmetric** cryptography involves the use of **two** keys:
  - a **public-key**, which may be known by anybody, and can be used to **encrypt messages**, and **verify signatures**
  - a **private-key**, known only to the recipient, used to **decrypt messages**, and **sign** (create) **signatures**
- is **asymmetric** because
  - those who encrypt messages or verify signatures **cannot** decrypt messages or create signatures

Why Public-Key Cryptography?

- developed to address two key issues:
  - **key distribution** – how to have secure communications in general without having to trust a KDC with your key
  - **digital signatures** – how to verify a message comes intact from the claimed sender

Public-Key Characteristics

- Public-Key algorithms rely on two keys with the characteristics that it is:
  - computationally infeasible to find decryption key knowing only algorithm & encryption key
  - computationally easy to en/decrypt messages when the relevant (en/decrypt) key is known
  - either of the two related keys can be used for encryption, with the other used for decryption (in some schemes)

Public-Key Cryptosystems

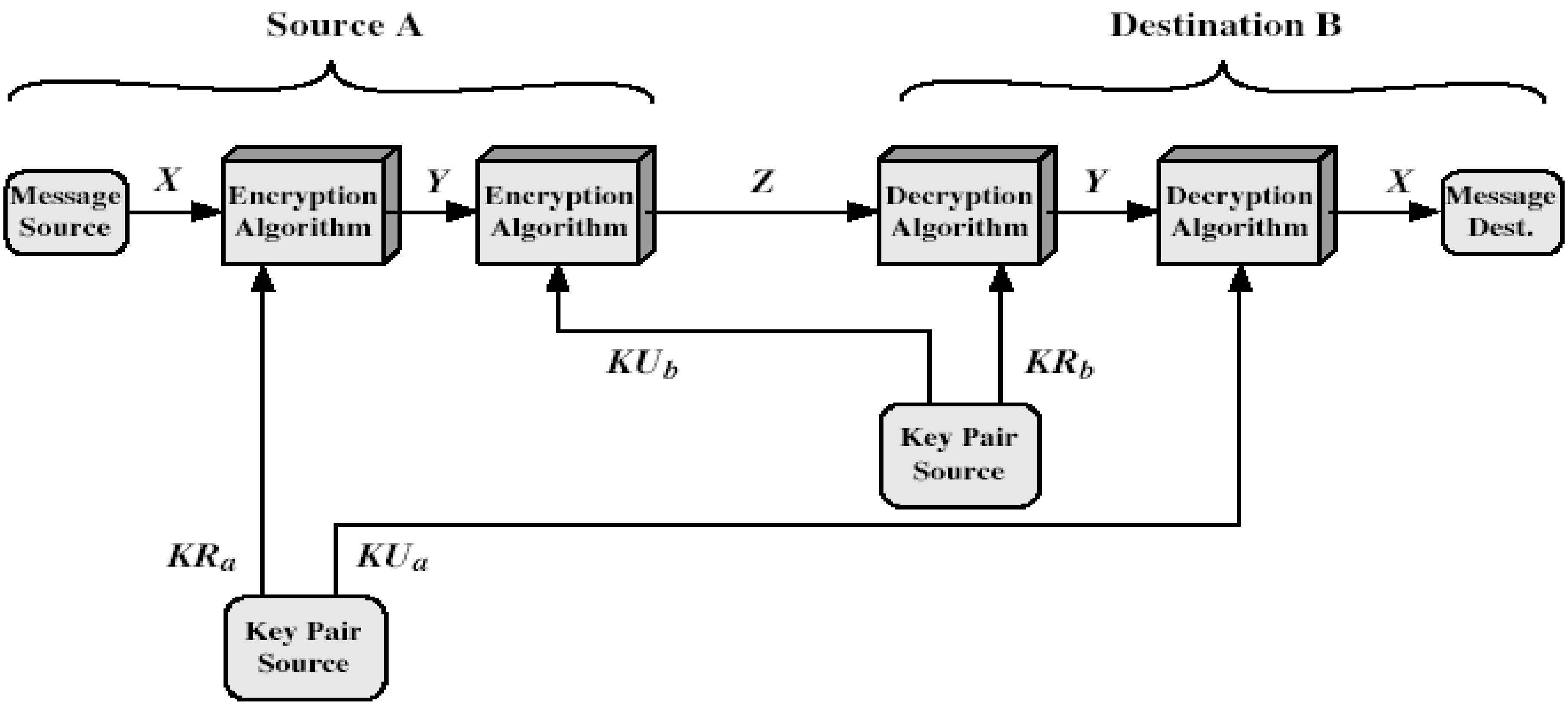


Figure 9.4 Public-Key Cryptosystem: Secrecy and Authentication

Security of Public Key Schemes

- like private key schemes brute force **exhaustive search** attack is always theoretically possible
- but keys used are too large (>512bits)
- security relies on a **large enough** difference in difficulty between **easy** (en/decrypt) and **hard** (cryptanalyse) problems
- requires the use of **very large numbers**
- hence is **slow** compared to private key schemes

RSA

- by Rivest, Shamir & Adleman of MIT in 1977
- best known & widely used public-key scheme
- based on exponentiation in a finite (Galois) field over integers modulo a prime
  - exponentiation takes $O((\log n)^3)$ operations (easy)
- uses large integers (eg. 1024 bits)
- security due to cost of factoring large numbers
  - factorization takes $O(e^{\log n \log n \log n})$ operations (hard)

RSA Key Setup

- each user generates a public/private key pair by:
- selecting two large primes at random - p, q
- computing their system modulus N=p.q
  - note $\phi(N)=(p-1)(q-1)$
- selecting at random the encryption key e

  (e that is relatively prime to $\phi(N)$)
  - where $1<e<\phi(N)$, $\gcd(e,\phi(N))=1$
- solve following equation to find decryption key d
  - $de \bmod \phi(N) = 1$ and $0 \le d \le N$
- publish their public encryption key: KU={e,N}
- keep secret private decryption key: KR={d,p,q}

ø(N) : (Euler function) number of positive integers less than n and relatively prime(서로소) to n

gcd : greatest common divisor (최대공약수)

RSA Use

- to encrypt a message M the sender:
  - obtains **public key** of recipient KU={e,N}
  - computes: $C=M^e \bmod N$, where $0 \le M < N$
- to decrypt the ciphertext C the owner:
  - uses their private key KR={d,p,q}
  - computes: $M=C^d \bmod N$
- note that the message M must be smaller than the modulus N (block if needed)

RSA Example

1. Select primes: $p=17$ & $q=11$
2. Compute $n = pq = 17 \times 11 = 187$
3. Compute $\varnothing(n) = (p-1)(q-1) = 16 \times 10 = 160$
4. Select e : $\gcd(e,160)=1$; choose $e=7$
5. Determine d: $1 = de \bmod 160$ and $d < 160$ Value is $d=23$ since $23 \times 7 = 161 = 1 \times 160 + 1$
6. Publish public key KU={7,187}
7. Keep secret private key KR={23,17,11}

RSA Example cont.

- sample RSA encryption/decryption is:
- given message $M = 88$ (nb. $88<187$)
- encryption:

$$C = 88^7 \bmod 187 = 11$$

- decryption:

$$M = 11^{23} \bmod 187 = 88$$

nb : **nota bene (유의하라 : note well)**

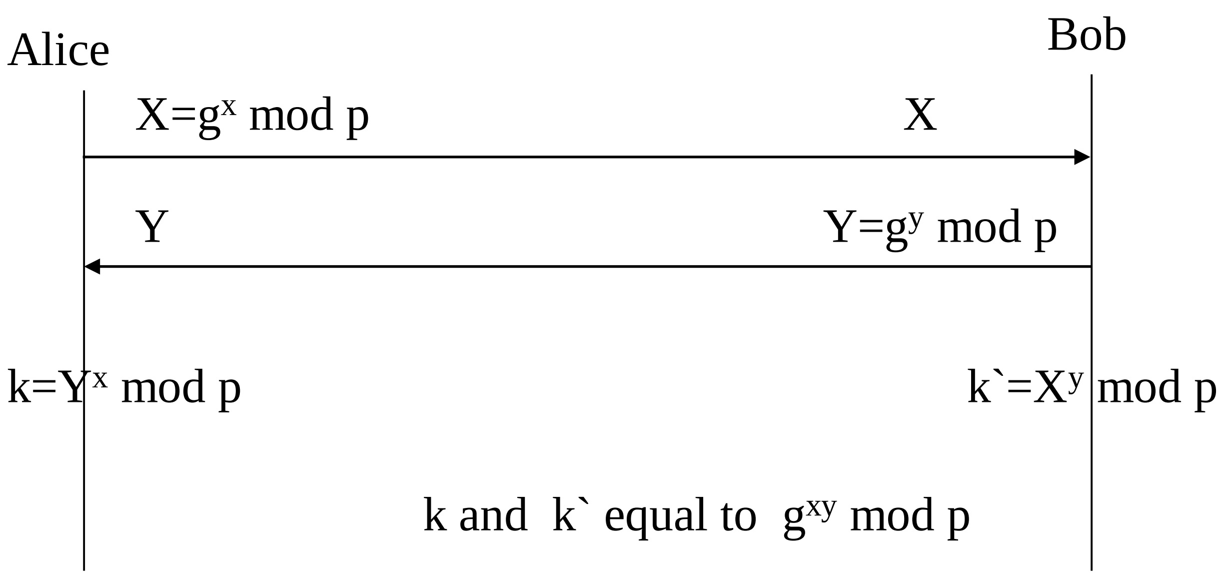
RSA Key Generation

- users of RSA must:
  - determine two primes at random - `p, q`
  - select either `e` or `d` and compute the other
- primes `p,q` must not be easily derived from modulus `N=p.q`
  - means must be sufficiently large
  - typically guess and use probabilistic test
- exponents `e,d` are inverses, so use Inverse algorithm to compute the other

Diffie-Hellman

- Key Distribution

p, g : large prime , g는 prime number p의 primitive root



Diffie-Hellman

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

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

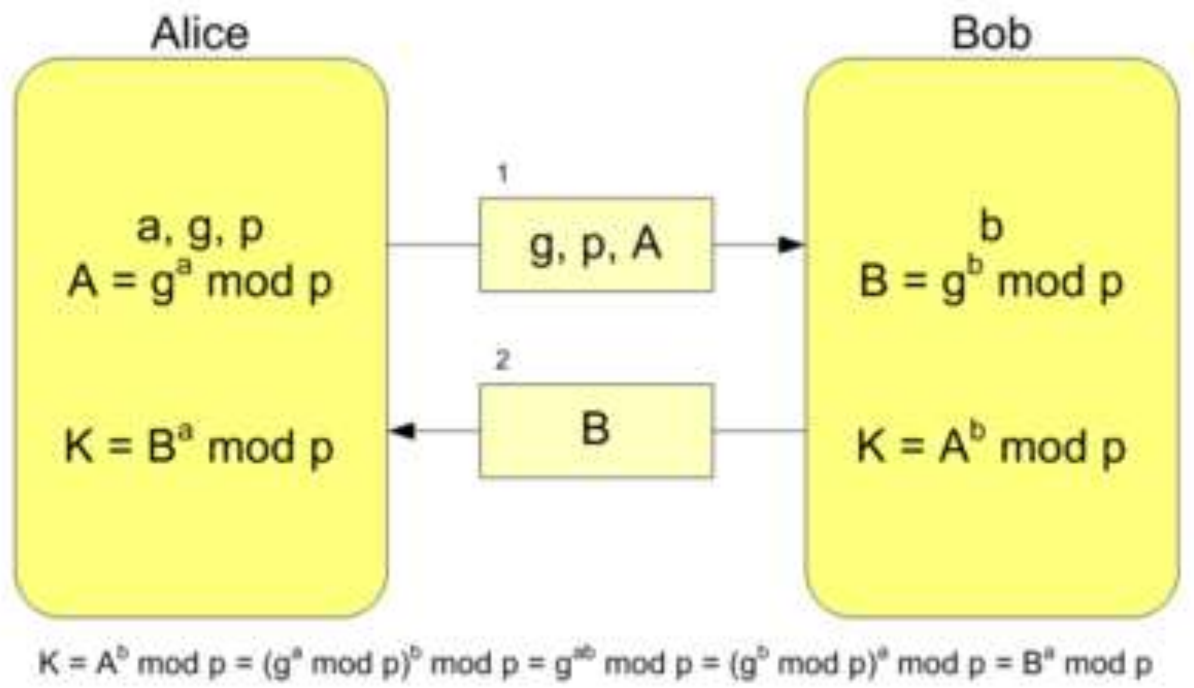
User B Key Generation	
Select private X_b	$X_b < q$
Calculate public Y_b	$Y_b = \alpha^{X_b} \bmod q$

Generation of Secret Key by User A
$K = (Y_b)^{X_a} \bmod q$

Generation of Secret Key by User B
$K = (Y_a)^{X_b} \bmod q$

Diffie-Hellman

Diffie-Hellman



Diffie-Hellman

Alice				Bob		
Secret	Public	Calculates	Sends	Calculates	Public	Secret
a	p, g		p,g$\longrightarrow$			b
a	p, g, A	$g^a \bmod p = A$	A$\longrightarrow$		p, g	b
a	p, g, A		$\longleftarrow$ B	$g^b \bmod p = B$	p, g, A, B	b
a, **s**	p, g, A, B	$B^a \bmod p = s$		$A^b \bmod p = s$	p, g, A, B	b, **s**

Diffie Hellman Key Exchange

	Alice	Evil Eve	Bob
	Alice and Bob exchange a Prime (P) and a Generator (G) in clear text, such that P > G and G is Primitive Root of P **G = 7, P = 11**	**Evil Eve sees** **G = 7, P = 11**	**Alice and Bob exchange a Prime (P) and a Generator (G) in clear text, such that P > G and G is Primitive Root of P** **G = 7, P = 11**
Step 1	Alice generates a random number: X_A X_A=6 (Secret)		Bob generates a random number: X_B X_B=9 (Secret)
Step 2	$Y_A=G^{X_A}(\text{mod } P)$ $Y_A = 7^6 \ (\text{mod } 11)$ $Y_A = 4$		$Y_B=G^{X_B}(\text{mod } P)$ $Y_B = 7^9 \ (\text{mod } 11)$ $Y_B = 8$
Step 3	Alice receives $Y_B = 8$ in clear-text	Evil Eve sees $Y_A = 4$, $Y_B = 8$	Bob receives $Y_A = 4$ in clear-text
Step 4	Secret Key $=Y_B^{X_A}(\text{mod } P)$ Secret Key $= 8^6 \ (\text{mod } 11)$ **Secret Key = 3**		Secret Key $=Y_A^{X_B}(\text{mod } P)$ Secret Key $= 4^9 \ (\text{mod } 11)$ **Secret Key = 3**

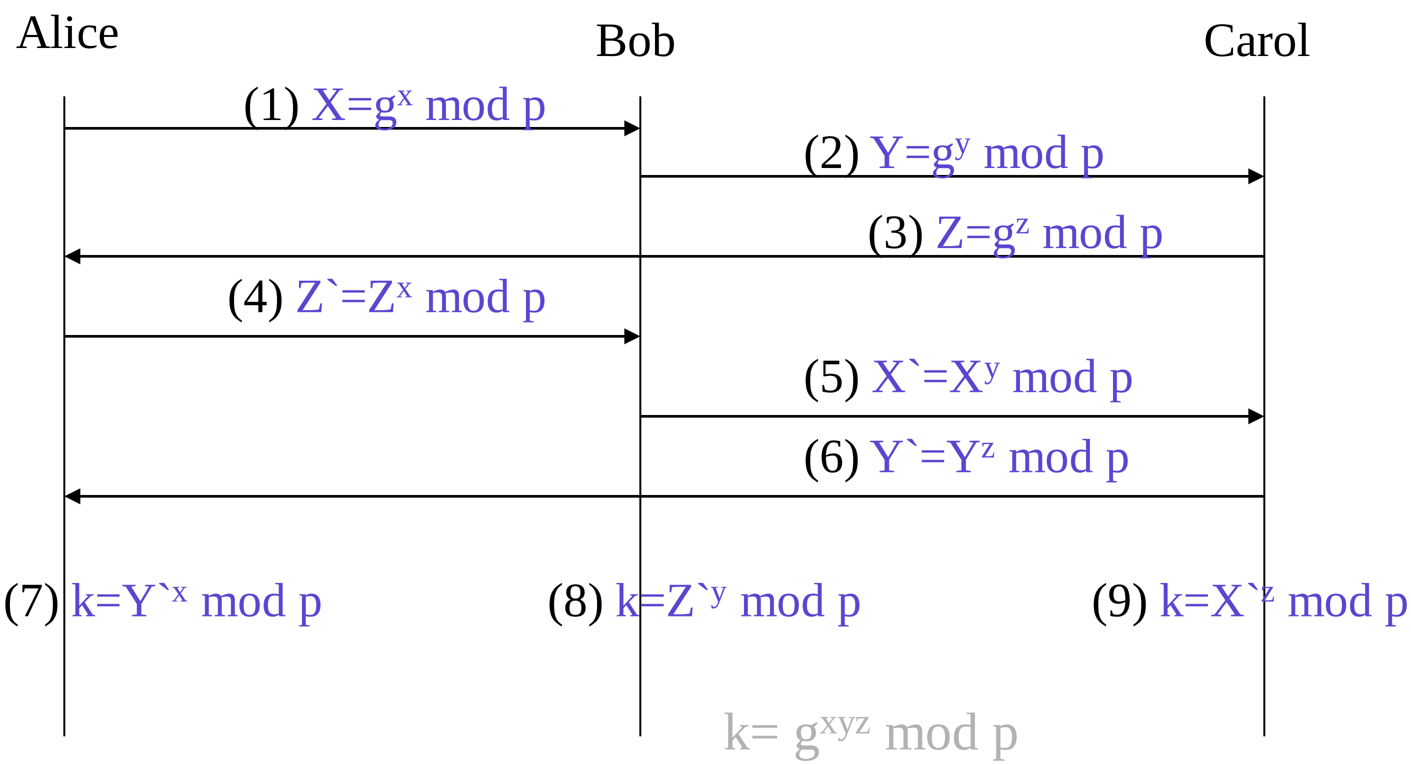
Diffie-Hellman

- Alice and Bob agree to use a prime number *p*=23 and base *g*=5.
- Alice chooses a secret integer *a*=6, then sends Bob (g^a mod *p*)
  – 5^6 mod 23 = 8.
- Bob chooses a secret integer *b*=15, then sends Alice (g^b mod p)
  – 5^{15} mod 23 = 19.
- Alice computes (g^b mod *p*)^a mod *p*
  – 19^6 mod 23 = 2.
- Bob computes (g^a mod *p*)^b mod *p*
  – 8^{15} mod 23 = 2.

base g : primitive root of p

Diffie-Hellman

- Diffie-Hellman with Three or More Parties
  - Alice, Bob, and Carol together generate a secret key.



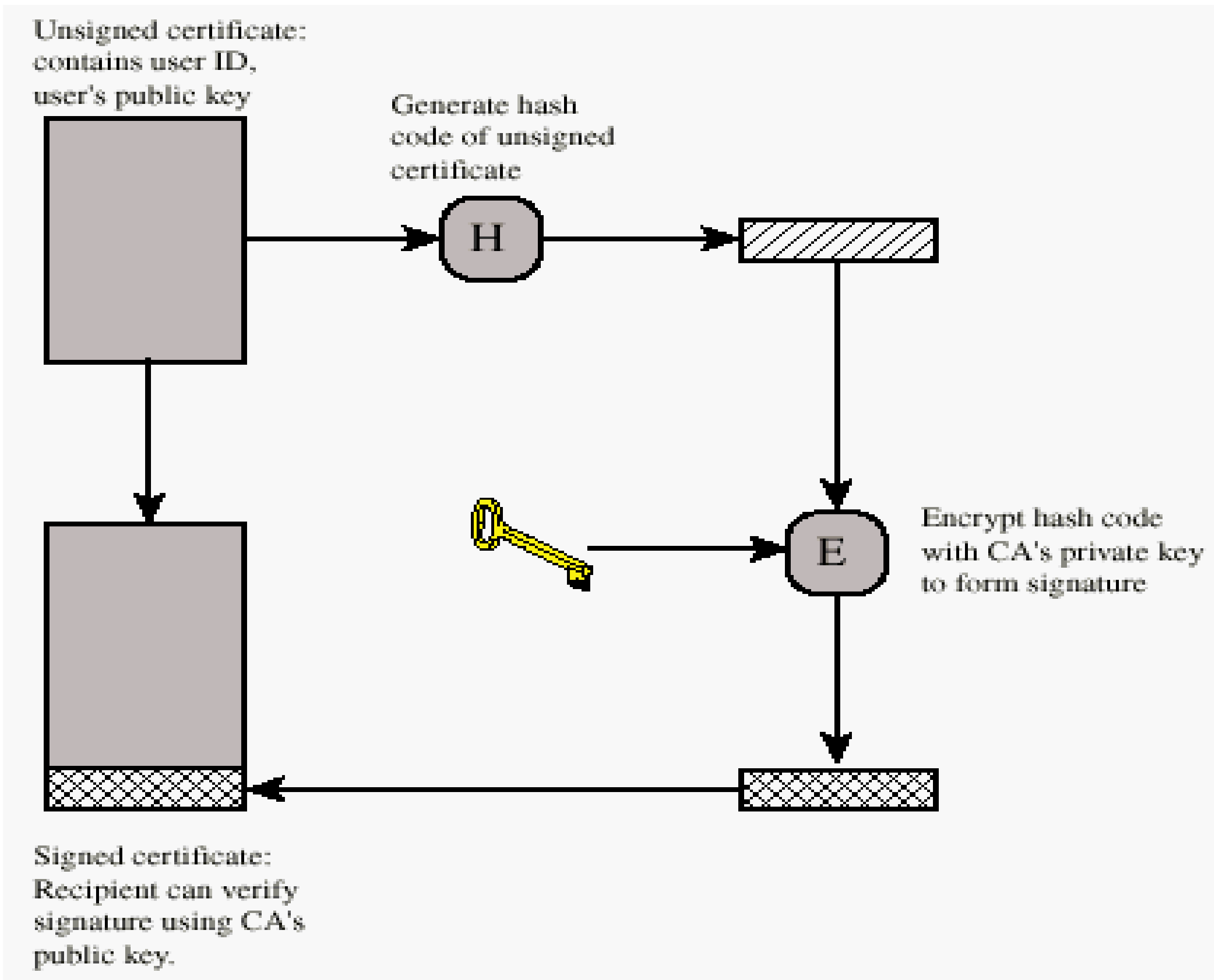
Diffie-Hellman

- Disadvantage of DF
  - Providing no authentication of the two communication partners

Other Public-Key Cryptographic Algorithms

- Digital Signature Standard (DSS)
  - Makes use of the SHA-1
  - Not for encryption or key echange
- Elliptic-Curve Cryptography (ECC)
  - Good for smaller bit size
  - Low confidence level, compared with RSA
  - Very complex

Key Management
– Public-Key Certificate Use



Summary

- Approaches to Message Authentication
  - MD5, SHA-1
- Secure Hash Functions and HMAC
- Public-Key Cryptography Principles
- Public-Key Cryptography Algorithms
  - RSA, Diffie Hellman
- Digital Signatures
- Key Management