Message Authentication

MAC and Hash

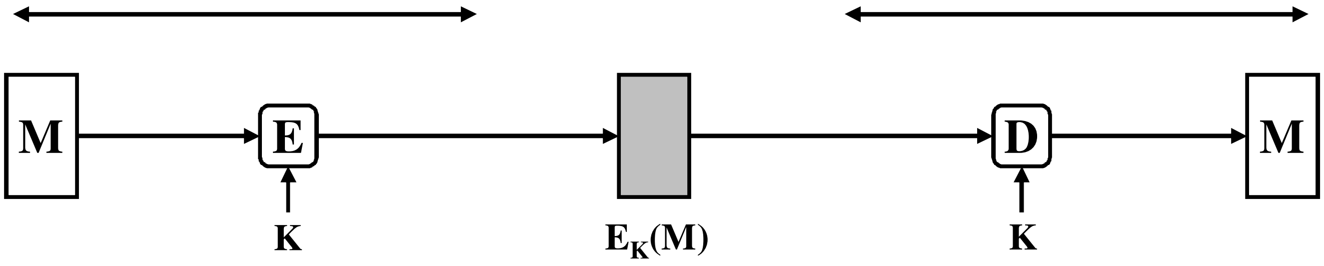
Message Authentication

- Verify that messages come from the alleged source, unaltered

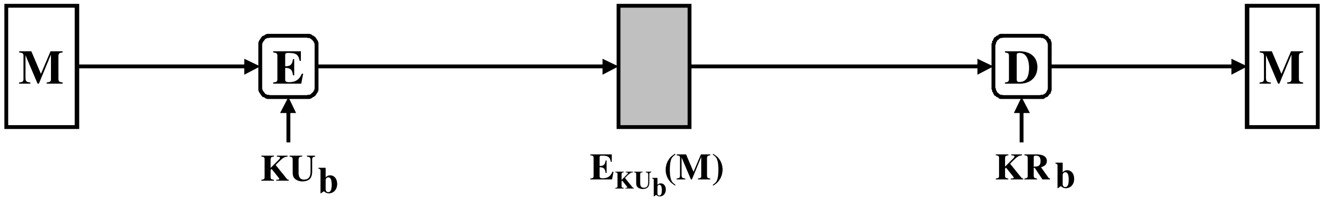
Authentication Functions

- Message encryption
  - Ciphertext itself serves as authenticator
- Message authentication code
  - Public function combines message and secret key into fixed length value
- Hash function
  - Public function maps message into fixed length value

Encryption for Authentication

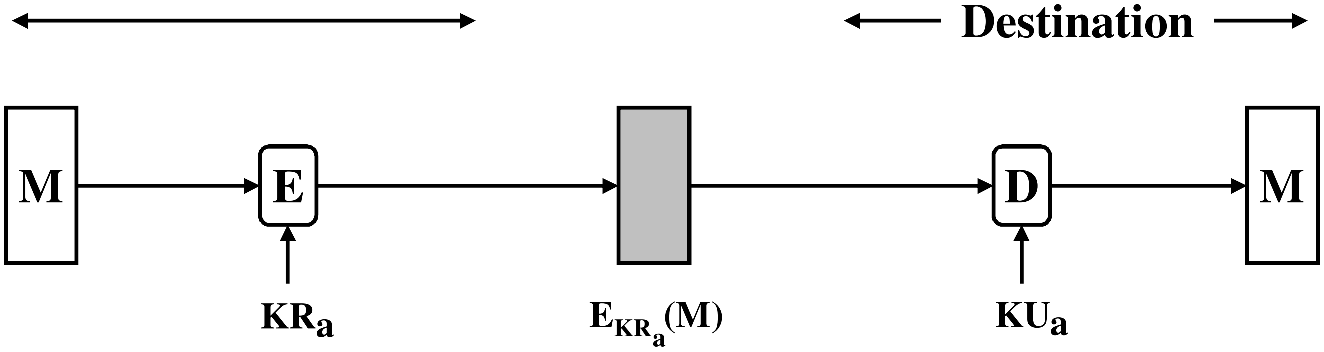


(a) Conventional encryption : confidentiality and authentication

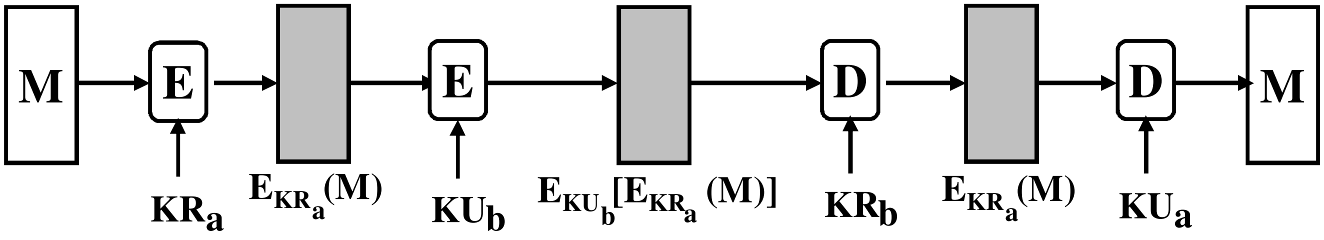


(b) Public-key encryption : confidentiality

Encryption for Authentication

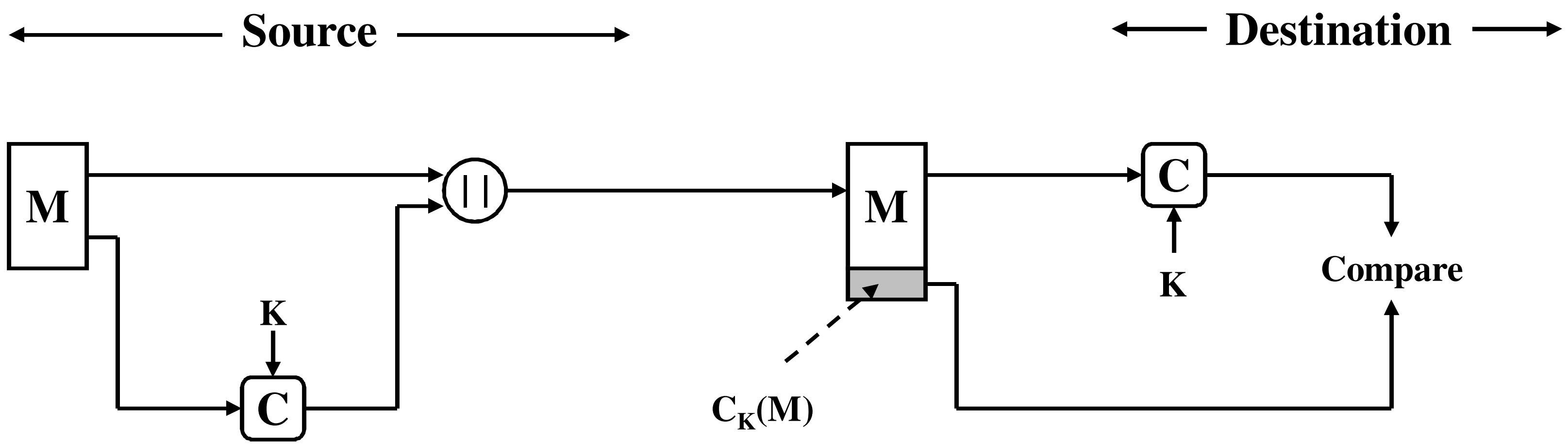


(c) Public-key encryption : authentication and signature

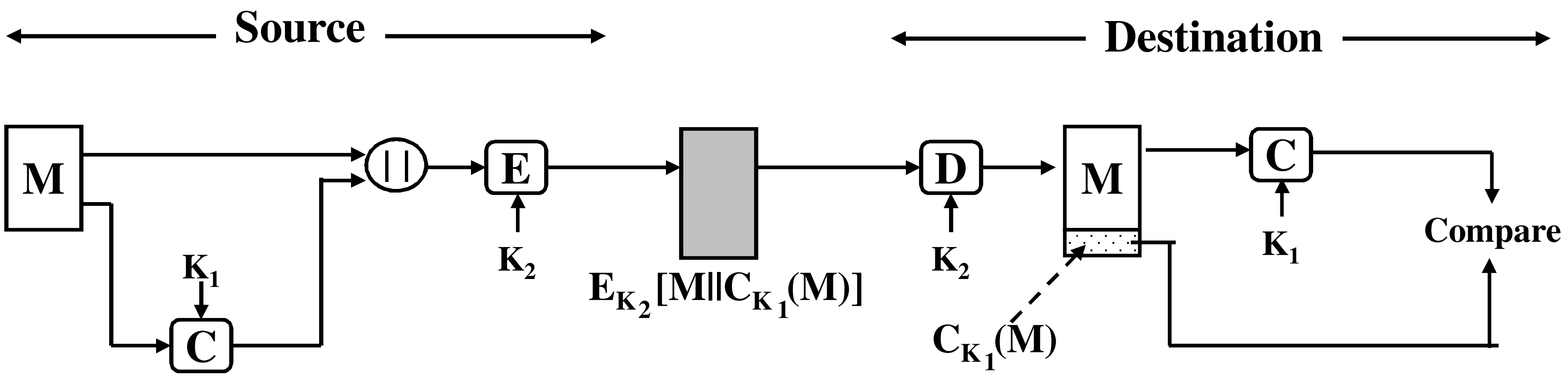


(d) Public-key encryption : confidentiality, authentication and signature

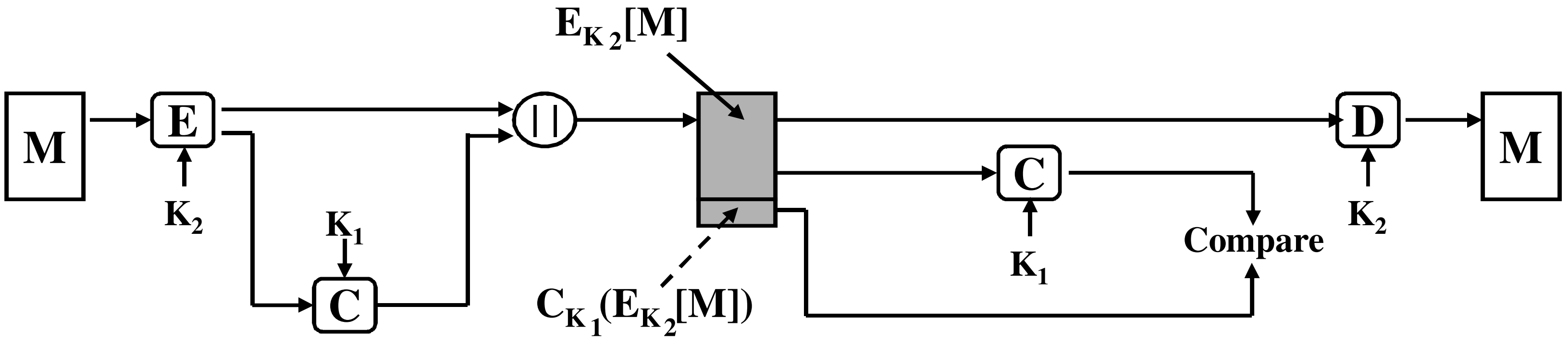
Message Authentication Code MAC



MAC (cont'd)



Message authentication and confidentiality; authentication tied to plaintext



Message authentication and confidentiality; authentication tied to ciphertext

Message Authentication Code MAC

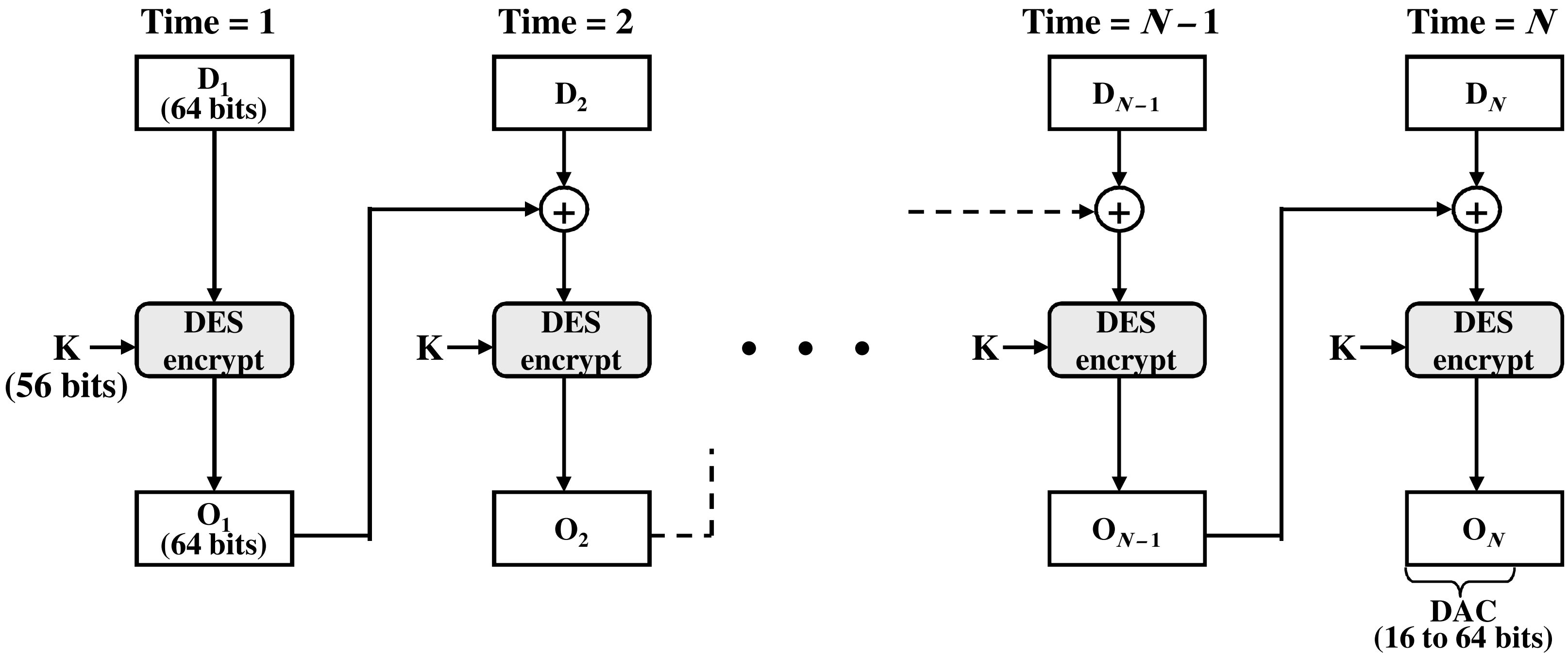
- Cryptographic checksum
- Mixes message with (shared) secret key to produce a fixed size block
- Assurances:
  - Message has not been altered
  - Message is from alleged sender
  - Message sequence is unaltered (requires internal sequencing)
- MAC algorithm need not be reversible

Why Use MACs?

  - Why not just use encryption?

- Clear-text stays clear
- MAC might be cheaper
- Broadcast
- Authentication of executables
- Separation of authentication check from message use

DES-Based MAC



MAC Requirements

- Given M and $C_k(M)$, it must be computationally infeasible to construct M′ s.t. $C_k(M) = C_k(M')$

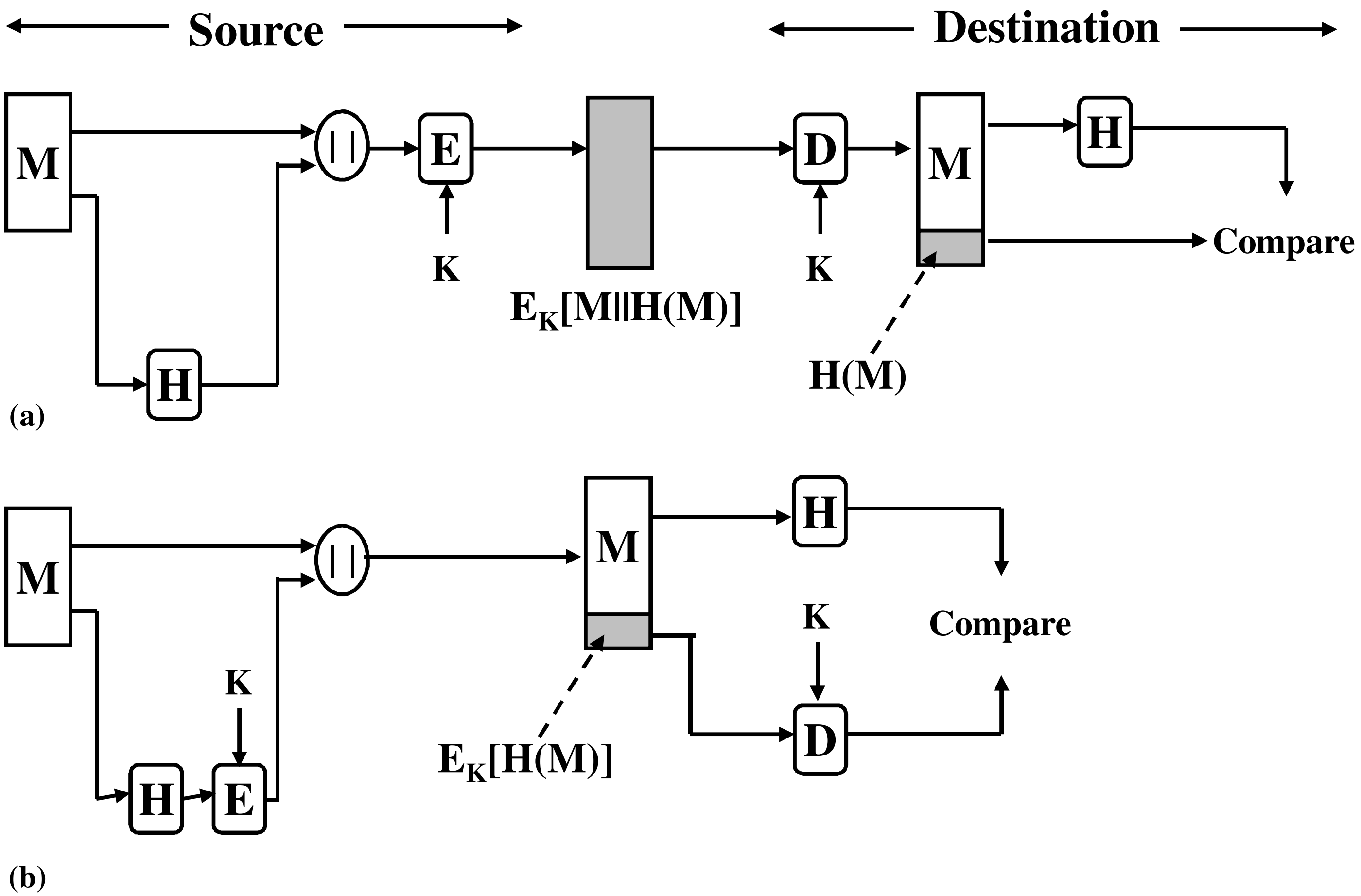
- Let M′ be equal to some known transformation on M. Then,

$$\Pr[C_k(M) = C_k(M')] = 2^{-n}.$$

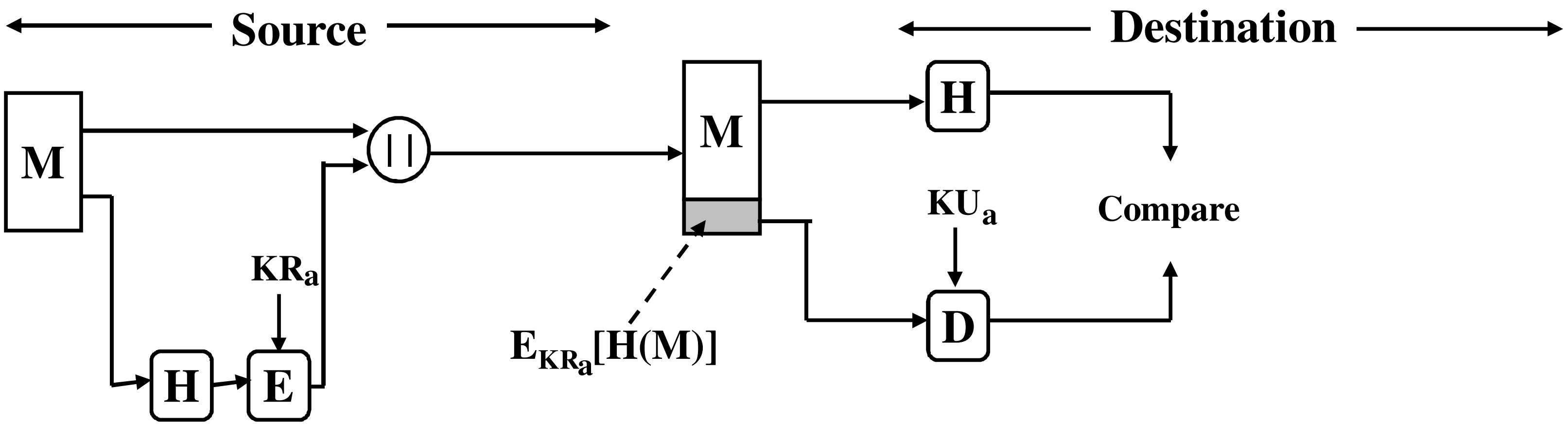
One-way Hash Functions

- Converts a variable size message M into fixed size hash code H(M)
- Can be used with encryption for authentication
  - E(M || H)
  - M || E(H)
  - M || signed H
  - E(M || signed H) gives confidentiality
  - M || H(M || K)
  - E(M || H(M || K))

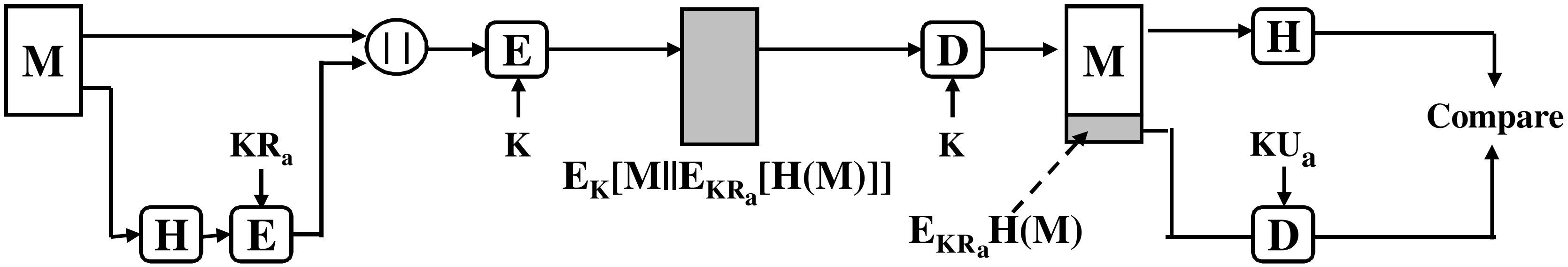
Hash (cont'd)


Source
Destination
M
||
E
K
$E_K[M||H(M)]$
D
K
M
H
Compare
H(M)
H
(a)
M
||
H
E
K
M
$E_K[H(M)]$
H
K
D
Compare
(b)

Hash (cont'd)

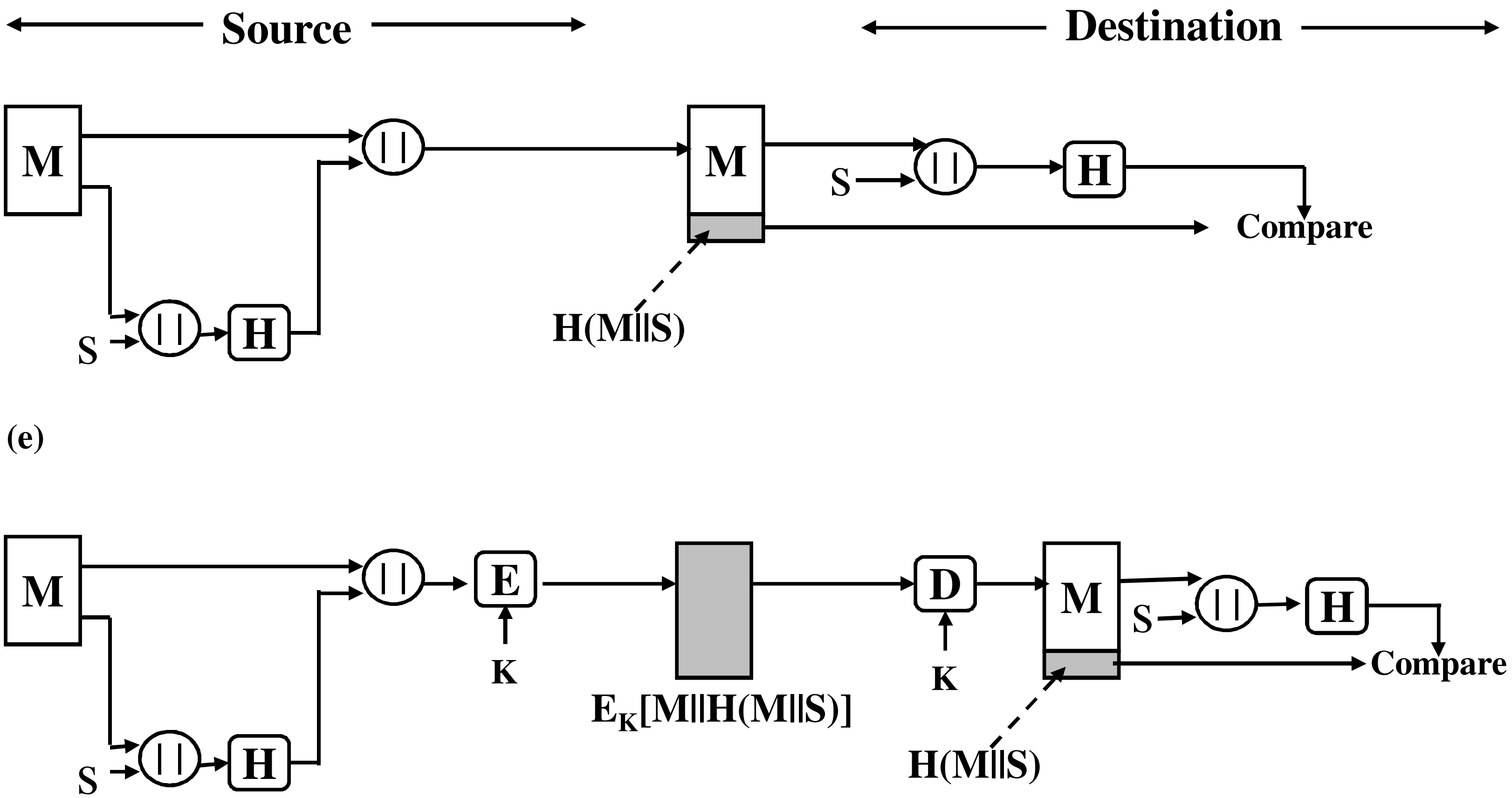


(c)



(d)

Hash (cont'd)


Source
Destination
M
S
| |
H
M
H(M||S)
S
| |
H
Compare
(e)
M
S
| |
H
| |
E
K
$E_K[M||H(M||S)]$
D
K
M
H(M||S)
S
| |
H
Compare
(f)

Hash Function Requirements

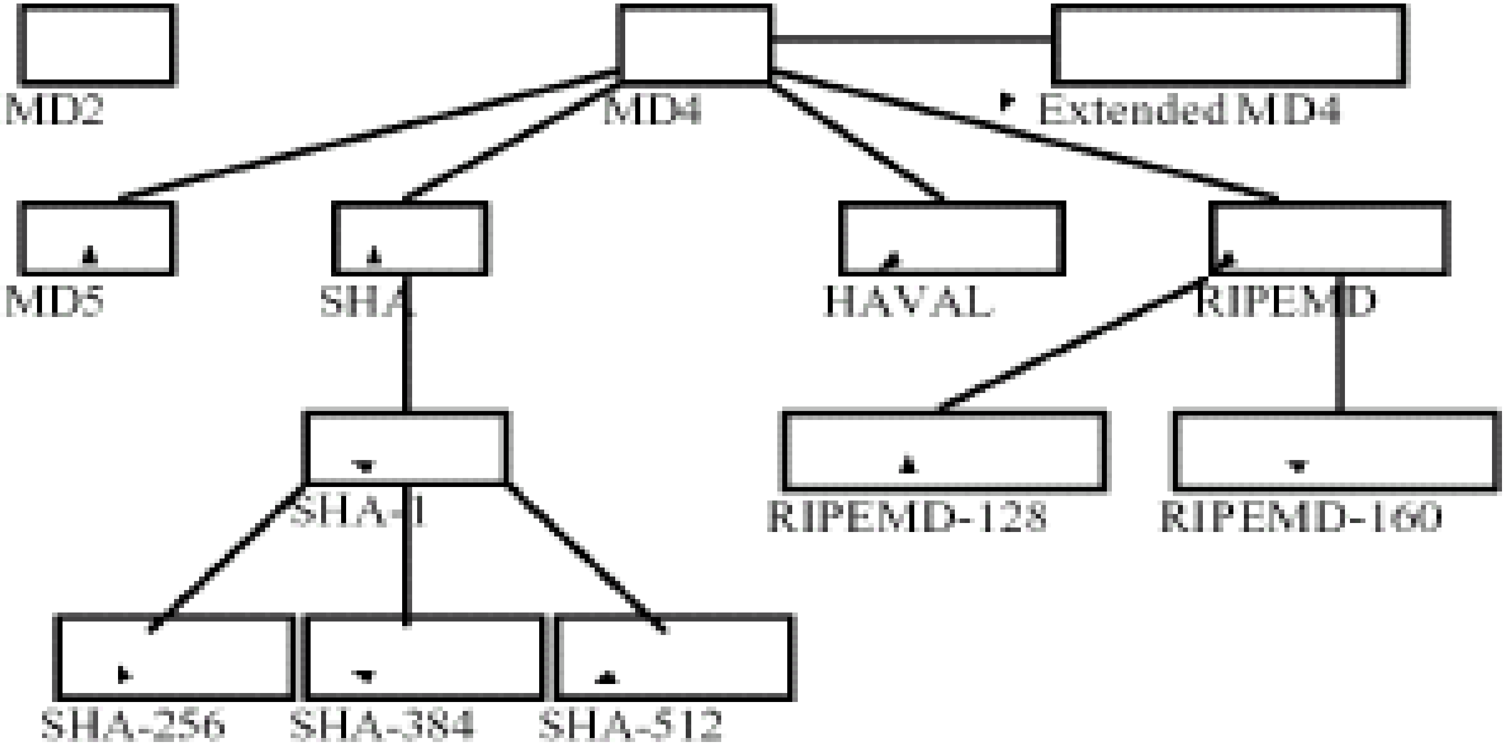
- H can be applied to any size data block
- H produces fixed length output
- H is fast
- H is *one-way*, i.e., given h, it is computationally infeasible to find any x s.t. h = H(x)

Cryptanalysis of Hash Functions

- General model of hash functions
  - Staged compression function f
  - L stages, Y_0, Y_1, …, Y_{L-1}
  - b input bits, n output bits per stage
  - initialization value
  - chaining variable
- $CV_0 = IV$
- $CV_i = f(Cv_{i-1}, Y_{i-1})$
- $H(M = Y_0 Y_1 \ldots Y_{L-1}) = CV_L$

Hash Algorithms

Popular Algorithms



MD5

- Message digest algorithm developed by Ron Rivest
- Algorithm takes a message of arbitrary length and produces a 128-bit digest
- The resulting digest is the unique "fingerprint" of the original message

Padding

- Message is padded so that its length in bits is congruent to 448 modulo 512
  - Length of padded message is 64 bits less than an integer multiple of 512 bits
- Padding is always added even if the message is the desired length
- Padding consists of a single 1 bit followed by 0 bits

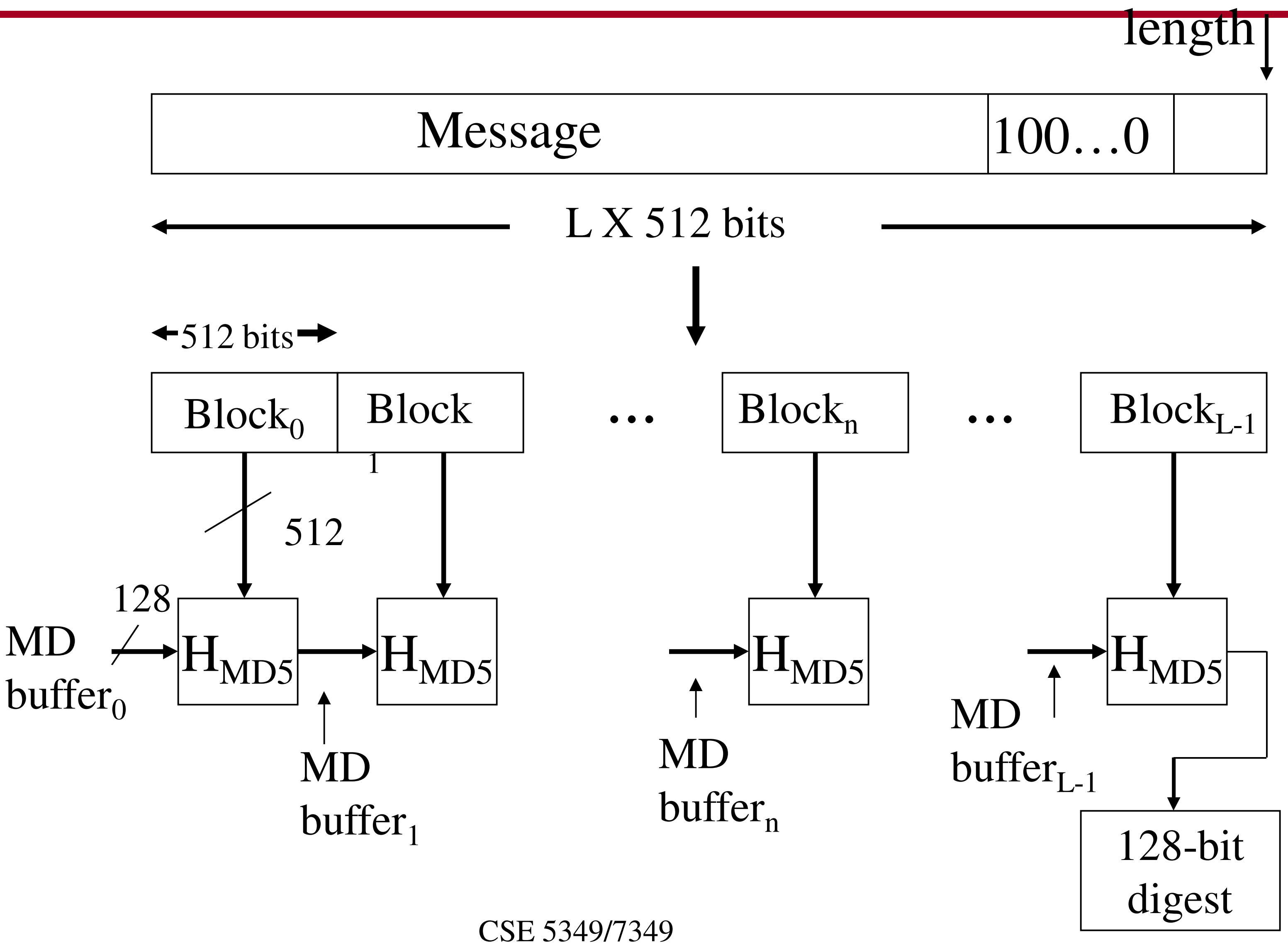
Append Length

- A 64 bit representation of the length in bits of the original message (before padding) is appended to the result of step 1
- If the original length is greater than 2^{64}, only the low-order 64 bits of the length are used
  - The length of the outcome of the first two steps is multiple of 512 bits

Initialize MD buffer

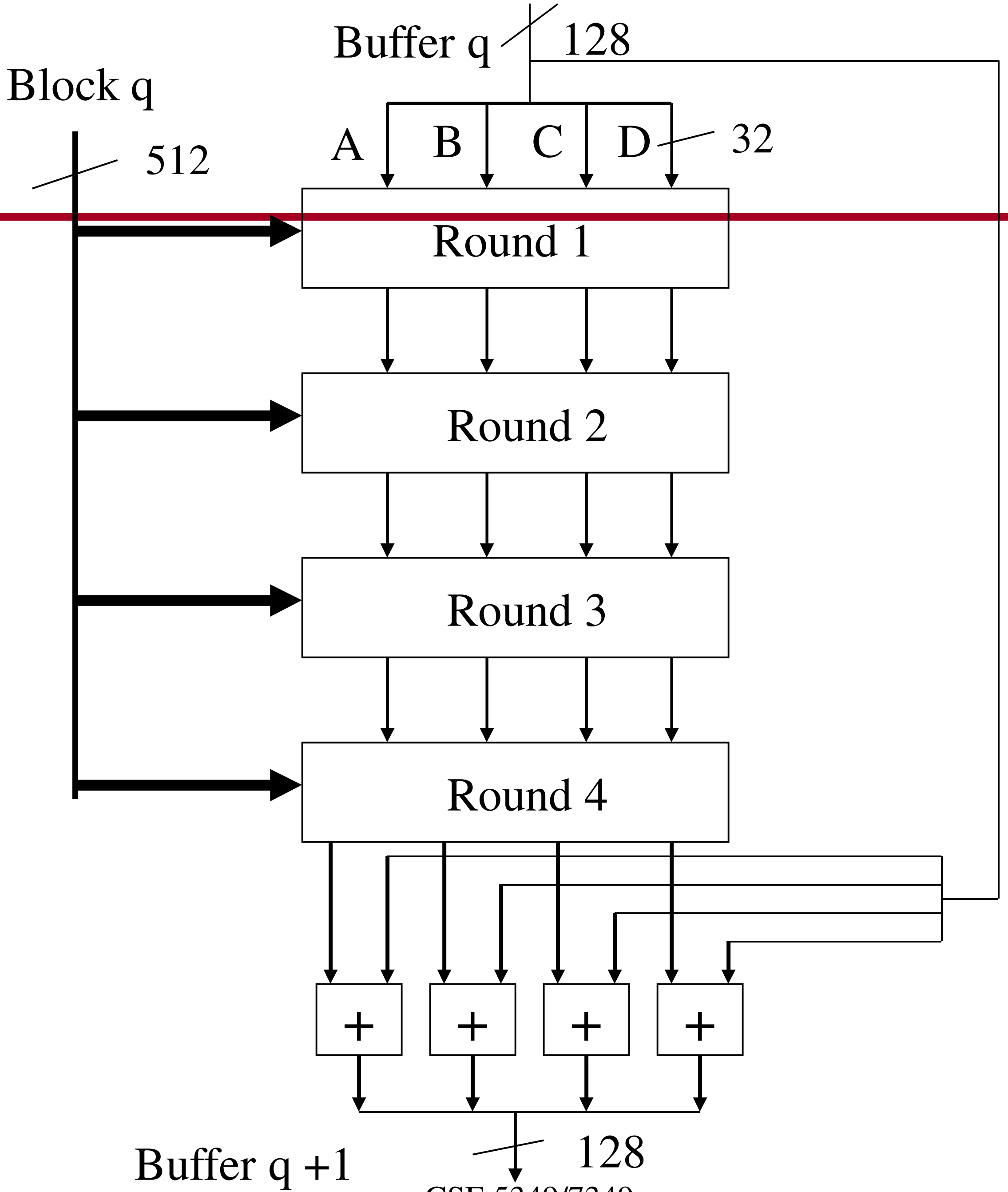
- A 128-bit buffer is used to hold intermediate and final results of the hash function
- Buffer can be represented as 4 32-bit registers (A,B,C,D)
- As 32 bit strings the init values (in hex):
  - word A: 01 23 45 67
  - word B: 89 AB CD EF
  - word C: FE DC BA 98
  - word D: 76 54 32 10

H_{MD5} = 4-round compression function

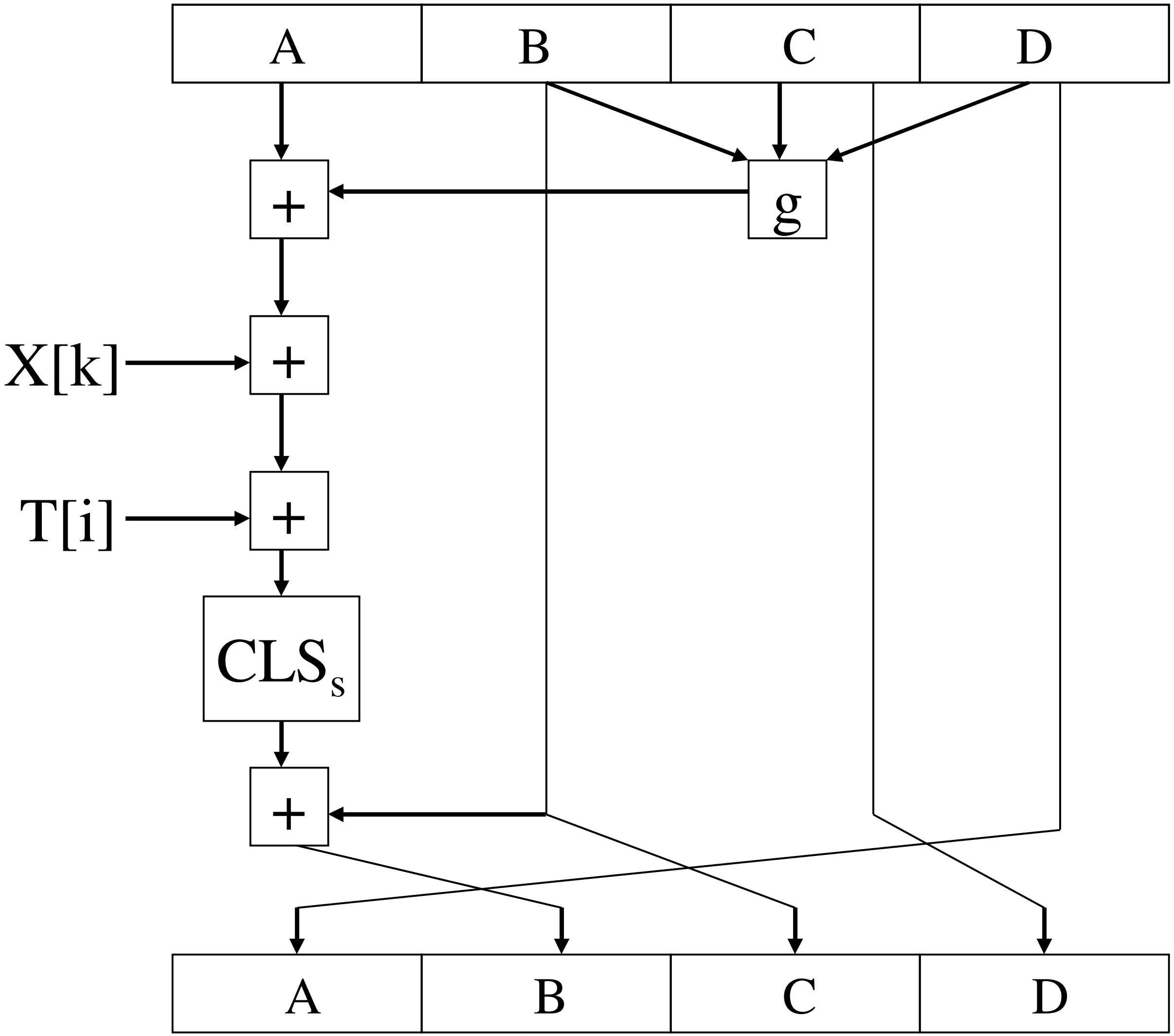


Message Processing

- Message is processed in 512-bit blocks
- Each block goes through a 4 round compression function
- After all 512-bit blocks have been processed, the output from the compression function is the 128-bit digest


Buffer q
128
Block q
512
A
B
C
D
32
Round 1
Round 2
Round 3
Round 4
+
+
+
+
Buffer q +1
128

- Each round is 16 steps, this is an ex.of a single step
- The order in which a,b,c,d is used produces a circular right shift of one word for each step



The Rounds

- $M_i=(w_0,\ldots,w_{15})$
- For fixed i, 4 consecutive steps will yield

$a_{i+4} = b_i + ((a_i + G_i\ (b_i,c_i,d_i)+w_i+t_i) \lll s_i)$

$d_{i+4}=a_i+((d_i+G_{i+1}\ (a_i,b_i,c_i)+w_{i+1}+t_{i+1}) \lll s_{i+1})$

$c_{i+4}=d_i+((c_i+G_{i+2}\ (d_i,a_i,b_i)+w_{i+2}+t_{i+2}) \lll s_{i+2})$

$b_{i+4}=c_i+((b_i+G_{i+3}\ (c_i,d_i,a_i)+w_{i+3}+t_{i+3}) \lll s_{i+3})$

t_i and s_i are predefined step dependant constants

$CLS_{s\ =si}$

- g = primitive function
- X[k] = kth 32-bit word in one of the 512 bit blocks
- T[i] = 2^{32} x abs(sin(i))
- Round 1
  - g(b,c,d) = (b AND c) OR (NOT b AND d)
  - k = 0…15
  - i = 1…16
- Round 2
  - g(b,c,d) = (b AND d) OR (c AND NOT d)
  - k = (1 + 5j)mod 16 where j = 1…16
  - i = 17..32

- Round 3
  - g(b,c,d) = b XOR c XOR d
  - k = (5 + 3j)mod 16 where j = 1…16
  - i = 33…48
- Round 4
  - g(b,c,d) = c XOR (b OR NOT d)
  - k = 7j mod 16 where j = 1…16
  - i = 49…64

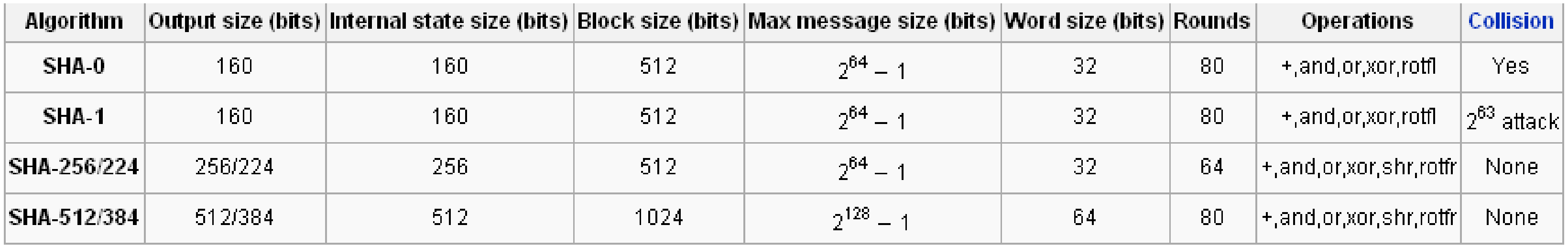
Some constants

M_j is the jth sub-block of the message block.
For step $i = 1$ *to* 64:
$t[i] = 2^{32} * \text{abs}(\sin(i))$ where i is measured in radians.

CLS_s is the number of bits to be shifted:
Round 1: [7, 12, 17, 22]
Round 2: [5, 9, 14, 20]
Round 3: [4, 11, 16, 23]
Round 4: [6, 10, 15, 21]

SHA1 & RIPEMD

SHA



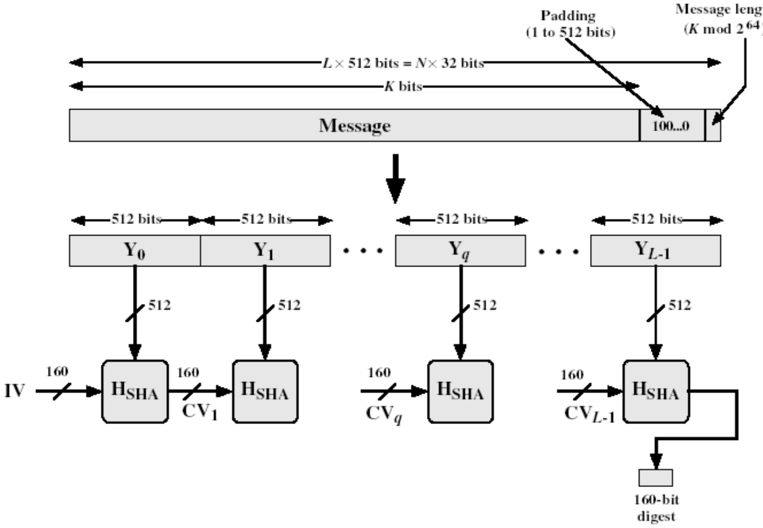
Algorithm	Output size (bits)	Internal state size (bits)	Block size (bits)	Max message size (bits)	Word size (bits)	Rounds	Operations	Collision
SHA-0	160	160	512	$2^{64} - 1$	32	80	+,and,or,xor,rotl	Yes
SHA-1	160	160	512	$2^{64} - 1$	32	80	+,and,or,xor,rotl	2^{63} attack
SHA-256/224	256/224	256	512	$2^{64} - 1$	32	64	+,and,or,xor,shr,rotr	None
SHA-512/384	512/384	512	1024	$2^{128} - 1$	64	80	+,and,or,xor,shr,rotr	None

Introduction

- Developed by NIST and published as FIP PUB 180 in 1993.
  - Revised version (SHA-1) issued as FIPS PUB 180-1 in 1995
- The algorithm takes as input a message with a maximum length of less than 2^{64} bits and produces a 160-bit message digest.
  - The input is processed in 512-bit blocks.

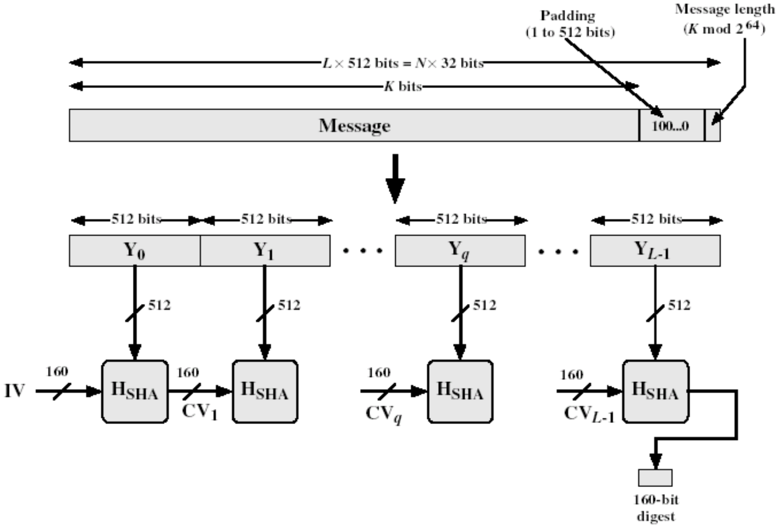
Message Extension

- The processing cycle consists of the following steps:
  - Append padding bits.
  - Append length.
  - Initialize MD buffer.
  - Process the plaintext message in 512 bit blocks.
  - Output the message digest for the plaintext message.



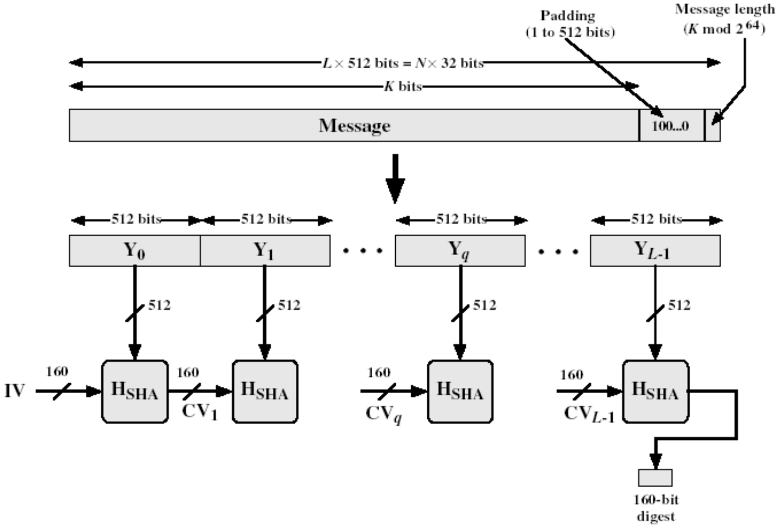
Message Extension (cont'd)

- In SHA-1 padding is always added to the plaintext message regardless of its length.
  - First append a binary "1", then as many binary "0"s as needed to make the padded message 64 bits short of a multiple of 512 bits.



Append Length

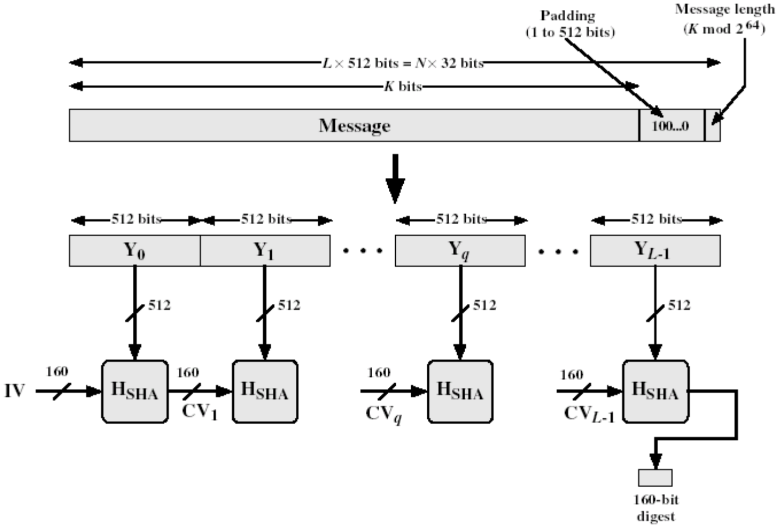
- Finally, a block of 64 bits is appended to the message.
  - It contains the length of the original plaintext message prior to padding.
  - This is an unsigned integer with the most significant bit (MSB) first.



Initialize MD Buffer

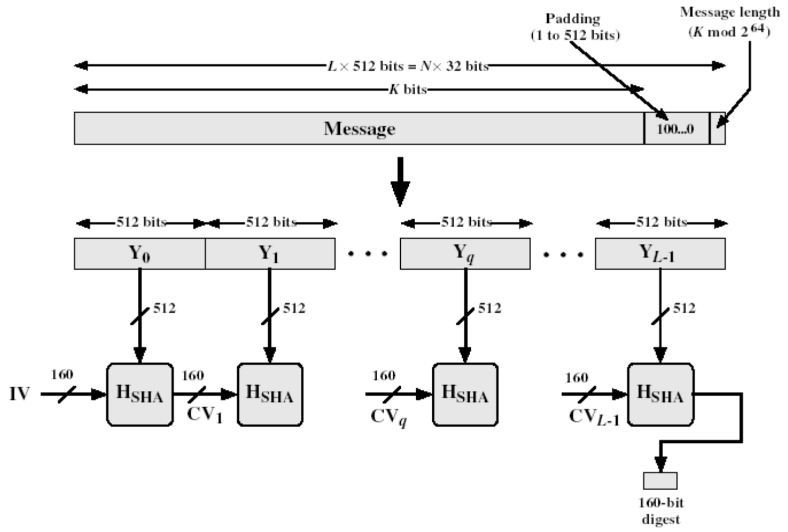
- A 160-bit buffer is used to hold intermediate and final results of the hash function.
  - It is represented as five 32-bit registers {A, B, C, D, E}.

- The initial register value are:
  - A = 67452301
  - B = EFCDAB89
  - C = 98BACDFE
  - D = 10325476
  - E = C3D2E1F0



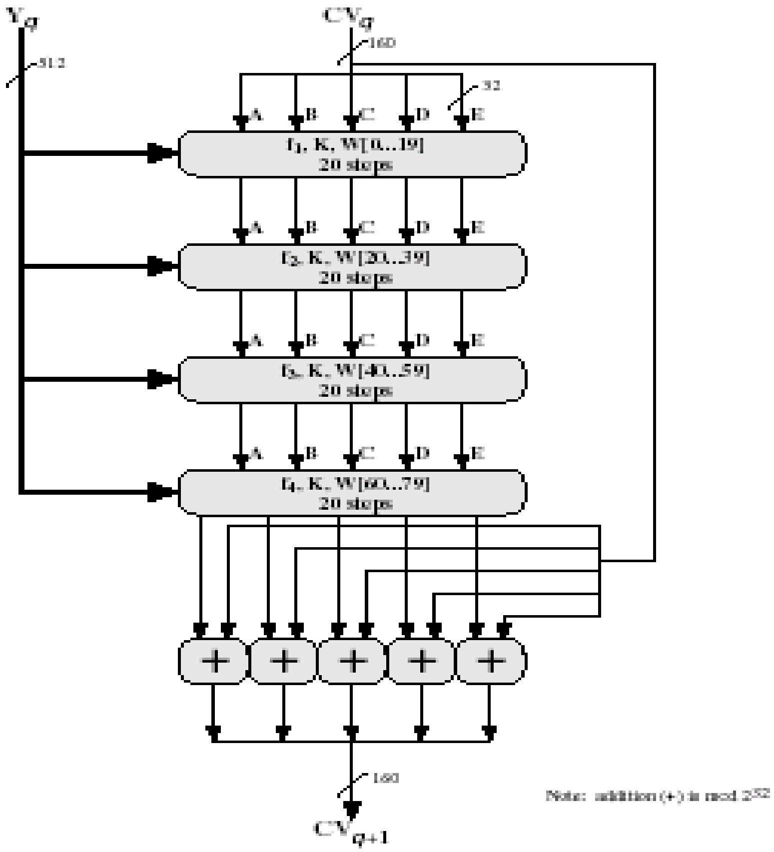
Message Processing

- The core of the algorithm is the H_{SHA} compression function that processes 512-bit blocks.



Message Processing (cont'd)

- The compression function consists of four rounds.
- Each round consists of 20 processing steps.
- The four rounds have a similar structure but each uses a different primitive logical function f_1, f_2, f_3, and f_4.



Note: addition (+) is mod 2^{32}

SHA-1
Primitive Functions (f_t)

Step Number	Function Name	Function Value
$0 \le t \le 19$	$f_1 = f(t, B, C, D)$	$(B \wedge C) \vee (\sim B \wedge D)$
$20 \le t \le 39$	$f_2 = f(t, B, C, D)$	$B \oplus C \oplus D$
$40 \le t \le 59$	$f_3 = f(t, B, C, D)$	$(B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$
$60 \le t \le 79$	$f_4 = f(t, B, C, D)$	$B \oplus C \oplus D$

Legend:

AND: $\wedge$
OR: $\vee$

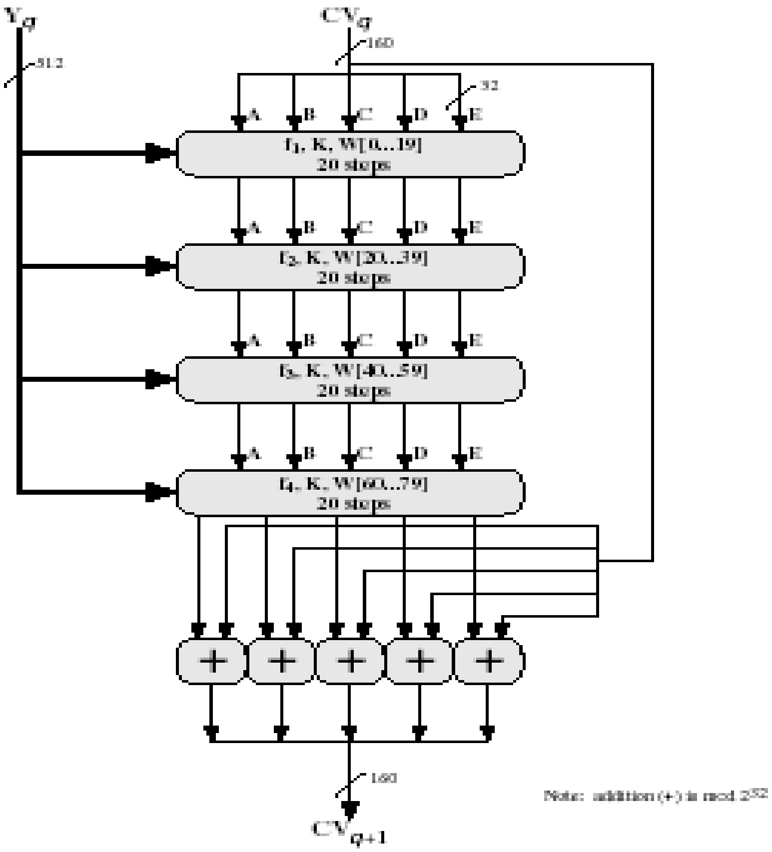
Not: $\sim$
XOR: $\oplus$

SHA-1
Truth Table for Function (f_t)

B	C	D	$f_{0\ldots19}$	$f_{20\ldots39}$	$f_{40\ldots59}$	$f_{60\ldots79}$
0	0	0	0	0	0	0
0	0	1	1	1	0	1
0	1	0	0	1	0	1
0	1	1	1	0	1	0
1	0	0	0	1	0	1
1	0	1	0	0	1	0
1	1	0	1	0	1	0
1	1	1	1	1	1	1

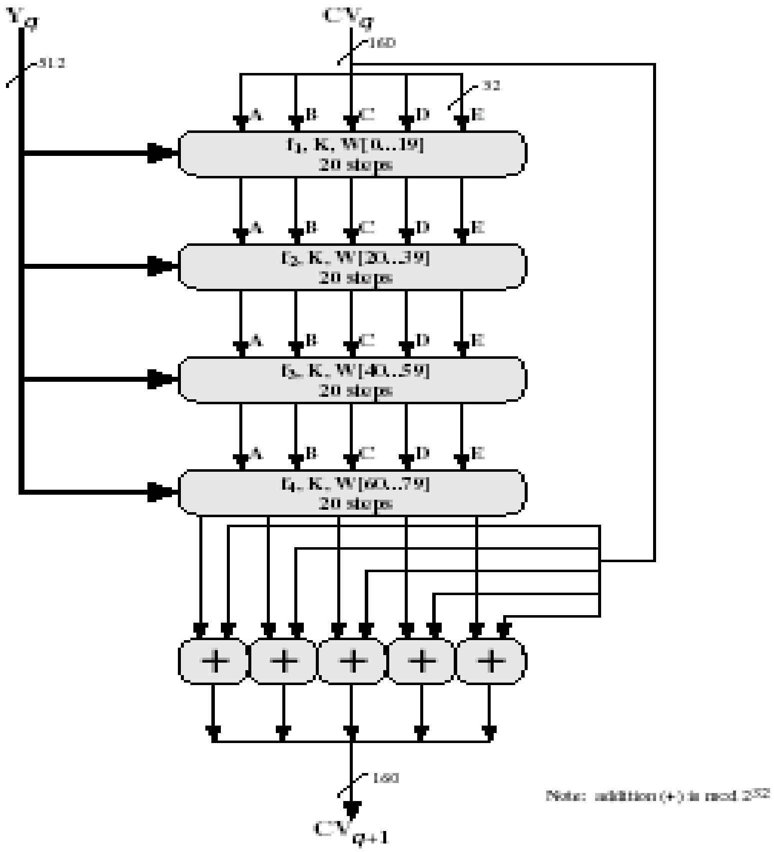
SHA-1 Secure Hash Function
512-bit Block Processing Function

- Each round takes as an input the current 512-bit block being processed Y_q and the 160-bit buffer value {ABCDE} and updates the contents of the buffer.
- Each round makes use of an additive constant K_t, where $0 \le t \le 79$ indicates one of 80 processing steps across four rounds.

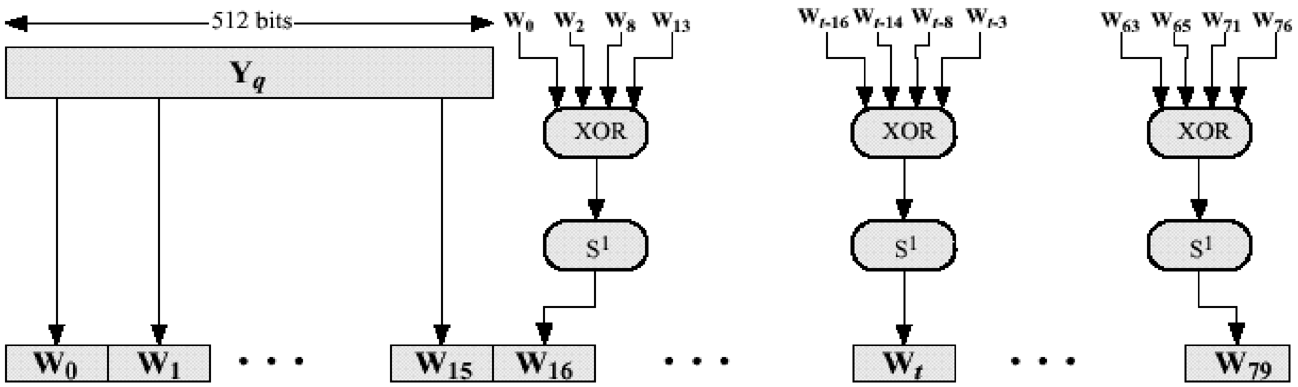


Additive Constants

- The value for these in hex are:
  - For 0 ≤ t ≤ 19
    - K_t = 5A827999
  - For 20 ≤ t ≤ 39
    - K_t = 6ED9EBA1
  - For 40 ≤ t ≤ 59
    - K_t = 8F1BBCDC
  - For 60 ≤ t ≤ 79
    - K_t = CA62C1D6



Deriving 32-bit Words (W_t)

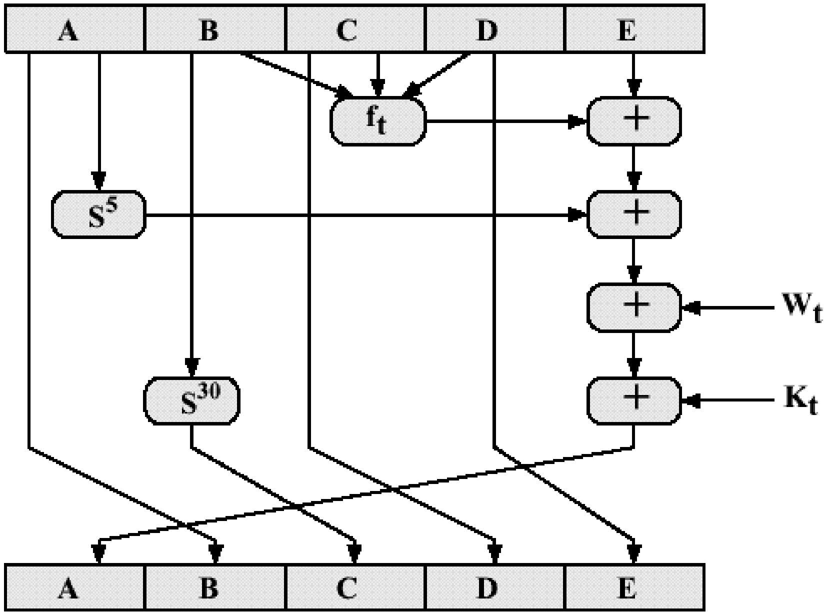


- The first sixteen values of W_t are taken directly from the 16 words of the current block and the remaining values are defined as …

$$W_t = W_{t-16} \oplus W_{t-14} \oplus W_{t-8} \oplus W_{t-3}$$

Single-step Operation

- The inputs to the step include:
  - The contents of Registers A to E respectively.
  - The additive constant K_t.
  - The constant W_t.
- (A,B,C,D,E) <- ((E+F(t,B,C,D)+(A<<5)+W_t+ K_t),A,(B<<30),C,D)



SHA-1 algorithm

```
Note: All variables are unsigned 32 bits and wrap modulo 232 when calculating

Initialize variables:
h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
h4 = 0xC3D2E1F0

Pre-processing:
append the bit '1' to the message
append k bits '0', where k is the minimum number ≥ 0 such that the resulting message
    length (in bits) is congruent to 448 (mod 512)
append length of message (before pre-processing), in bits, as 64-bit big-endian integer

Process the message in successive 512-bit chunks:
break message into 512-bit chunks
for each chunk
    break chunk into sixteen 32-bit big-endian words w[i], 0 <= i <= 15

    Extend the sixteen 32-bit words into eighty 32-bit words:
    for i from 16 to 79
        w[i] = (w[i-3] xor w[i-8] xor w[i-14] xor w[i-16]) leftrotate 1

    Initialize hash value for this chunk:
    a = h0
    b = h1
    c = h2
    d = h3
    e = h4
….
```

SHA-1 algorithm

```
Main loop:
  for i from 0 to 79
     if $0 \le i \le 19$ then
        f = (b and c) or ((not b) and d)
        k = 0x5A827999
     else if $20 \le i \le 39$
        f = b xor c xor d
        k = 0x6ED9EBA1
     else if $40 \le i \le 59$
        f = (b and c) or (b and d) or (c and d)
        k = 0x8F1BBCDC
     else if $60 \le i \le 79$
        f = b xor c xor d
        k = 0xCA62C1D6

     temp = (a leftrotate 5) + f + e + k + w[i]
     e = d
     d = c
     c = b leftrotate 30
     b = a
     a = temp

  Add this chunk's hash to result so far:
  h0 = h0 + a
  h1 = h1 + b
  h2 = h2 + c
  h3 = h3 + d
  h4 = h4 + e

Produce the final hash value (big-endian):
digest = hash = h0 append h1 append h2 append h3 append h4
```

SHA-1 vs. MD5

- Security against brute-force attacks
  - 32 bits longer than the MD5
  - Producing any message having a given message digest is on the order 2^{160} for SHA-1
  - Producing 2 messages having the same message digest is on the order 2^{80} for SHA-1
  - Stronger against brute-force attack

Comparison (cont'd)

- Security against cryptanalysis
  - Less vulnerable against cryptanalytic attacks discovered since MD5's design
- Speed
  - Both algorithms rely heavily on addition modulo 2^{32} SHA-1 involves more steps and must process a 160-bit buffer.
  - SHA-1 should be slower than MD5

Comparison (cont'd)

- Simplicity and Compactness
  - Both are simple to describe and simple to implement
  - Do not require large programs nor substitution tables
- Little-endian vs Big-endian architecture
  - There appears to be no advantage to either approach

RIPEMD-160

- Developed under the European RACE Integrity Primitives Evaluation project
- By a group of researchers launching partially successful attacks on MD4 and MD5
- Originally a 128-bit RIPEMD

RIPEMD-160 Logic

- INPUT: a message of arbitrary length
- Overall processing: Similar to MD5 with a block length of 512 bits and a hash length of 160 bits
- Output: 160-bit message digest

Processing Steps

1. Append padding bits
2. Append length
3. Initialize MD buffer
4. Process message in 512-bit blocks
5. Output

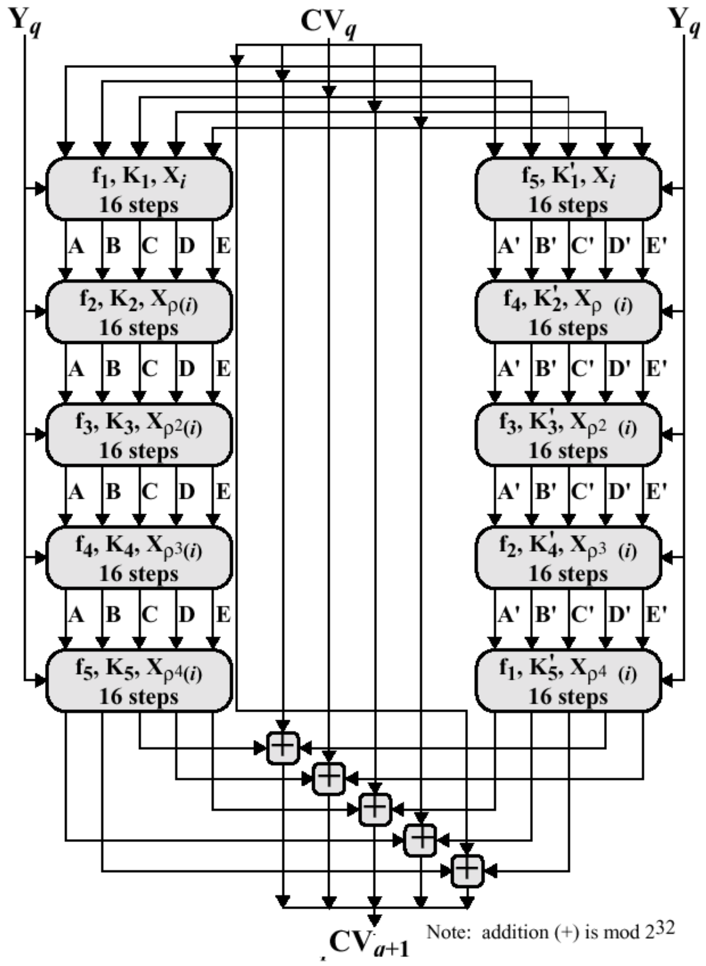
Processing (cont'd)

- Initialize MD buffer
  - 160-bit buffer
  - 5 32-bit registers (A, B, C, D, E)
  - IV = {A=67452301,B=EFCDAB89,C=98BADCFE,
    D=10325476,E=C3D2E1F0}
  - Stored in little-endian format

Message Processing

- Process message in 512bit blocks
  - Module that consists of 10 rounds of processing of 16 steps each
  - 10 rounds are arranged as 2 parallel lines of 5 rounds
  - 4 rounds have a similar structure, but each uses a different primitive logical function(f_1,f_2,f_3,f_4,f_5)
  - INPUT: 512-bit block Y_q, 160-bit CV_q ABCDE(L), A'B'C'D'E'(R)
  - Each round uses an additive 9 constants
  - OUTPUT: CV_{q+1} (addition is mod 2^{32})

Rounds



Note: addition (+) is mod 2^{32}

- $CV_{q+1}(0)=CV_q(1)+C+D'$
- $CV_{q+1}(1)=CV_q(2)+D+E'$
- $CV_{q+1}(2)=CV_q(3)+E+A'$
- $CV_{q+1}(3)=CV_q(4)+A+B'$
- $CV_{q+1}(4)=CV_q(0)+B+C'$

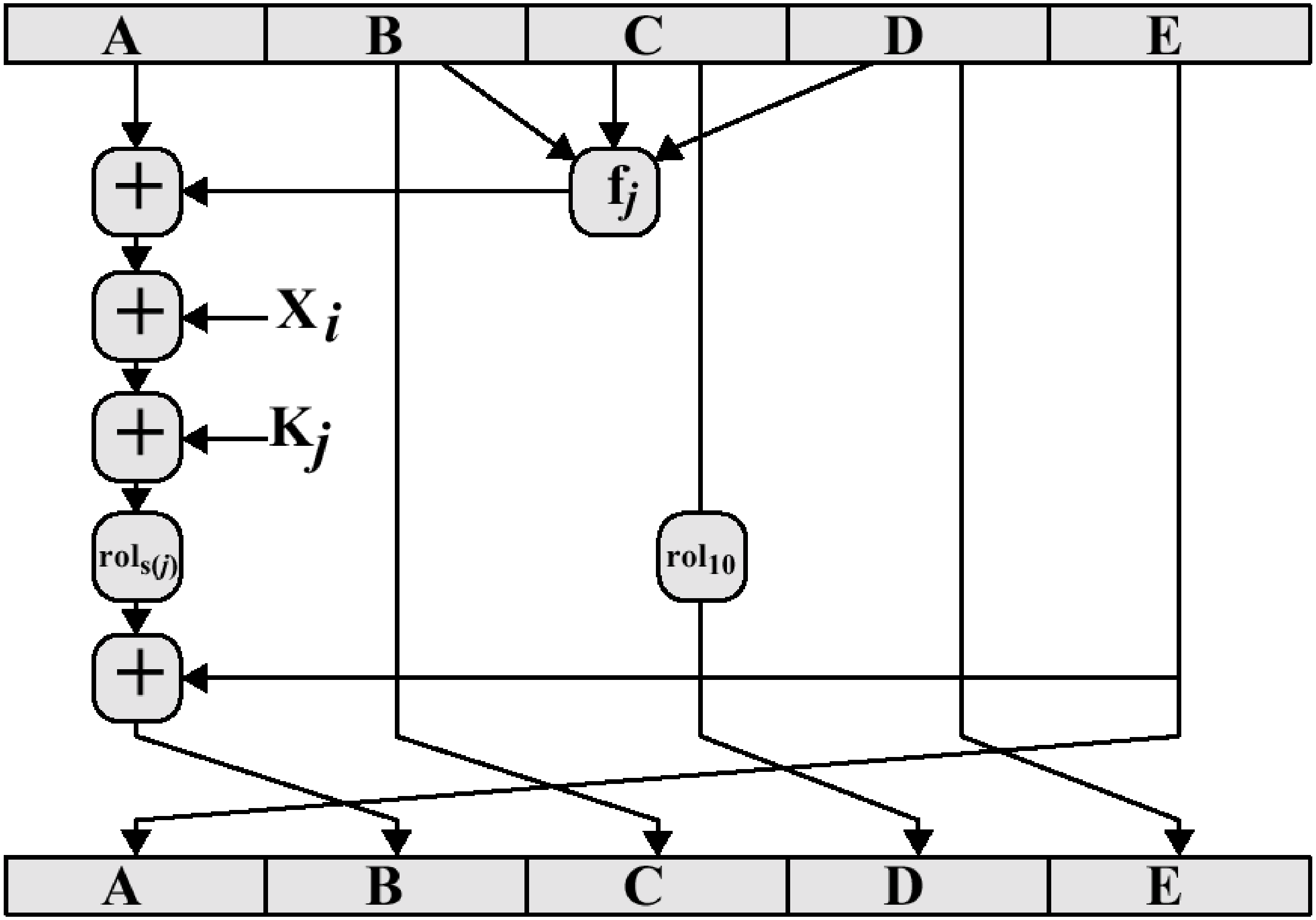
	Left Half		**Right Half**	
Step Number	**Hexadecimal**	**Integer part of:**	**Hexadecimal**	**Integer part of:**
$0 \le j \le 15$	$K_1 = K(j) =$ 00000000	0	$K'_1 = K'(j) =$ 50A28BE6	$2^{30} \times \sqrt[3]{2}$
$16 \le j \le 31$	$K_2 = K(j) =$ 5A827999	$2^{30} \times \sqrt{2}$	$K'_2 = K'(j) =$ 5C4DD124	$2^{30} \times \sqrt[3]{3}$
$32 \le j \le 47$	$K_3 = K(j) =$ 6ED9EBA1	$2^{30} \times \sqrt{3}$	$K'_3 = K'(j) =$ 6D703EF3	$2^{30} \times \sqrt[3]{5}$
$48 \le j \le 63$	$K_4 = K(j) =$ 8F1BBCDC	$2^{30} \times \sqrt{5}$	$K'_4 = K'(j) =$ 7A6D76E9	$2^{30} \times \sqrt[3]{7}$
$64 \le j \le 79$	$K_5 = K(j) =$ A953FD4E	$2^{30} \times \sqrt{7}$	$K'_5 = K'(j) =$ 00000000	0

Compression

- Each round consists of a sequence of 16 steps [Figure 9.9]
- The processing algorithm of one round

```
$A:=CV_q(0);B:=CV_q(1);C:=CV_q(2);D:=CV_q(3);E:= CV_q(4)$
$A':=CV_q(0);B':=CV_q(1);C':=CV_q(2);D':=CV_q(3);E':= CV_q(4)$
for j=0 to 79 do
  $T:=rol_{s(j)}(A+f(j,B,C,D)+X_{r(j)}+K(j))+E;$
  $A:=E;E:=D;D:= rol_{10}(C);C:=B;B:=T;$
  $T:=rol_{s'(j)}(A'+f(79-j,B',C',D')+X_{r'(j)}+K'(j))+E';$
  $A':=E';E':=D';D':= rol_{10}(C');C':=B';B':=T';$
enddo
$CV_{q+1}(0)=CV_q(1)+C+D';$  $CV_{q+1}(1)=CV_q(2)+D+E';$  $CV_{q+1}(2)=CV_q(3)+E+A';$
$CV_{q+1}(3)=CV_q(4)+A+B';$  $CV_{q+1}(4)=CV_q(0)+B+C';$
```

Single Step



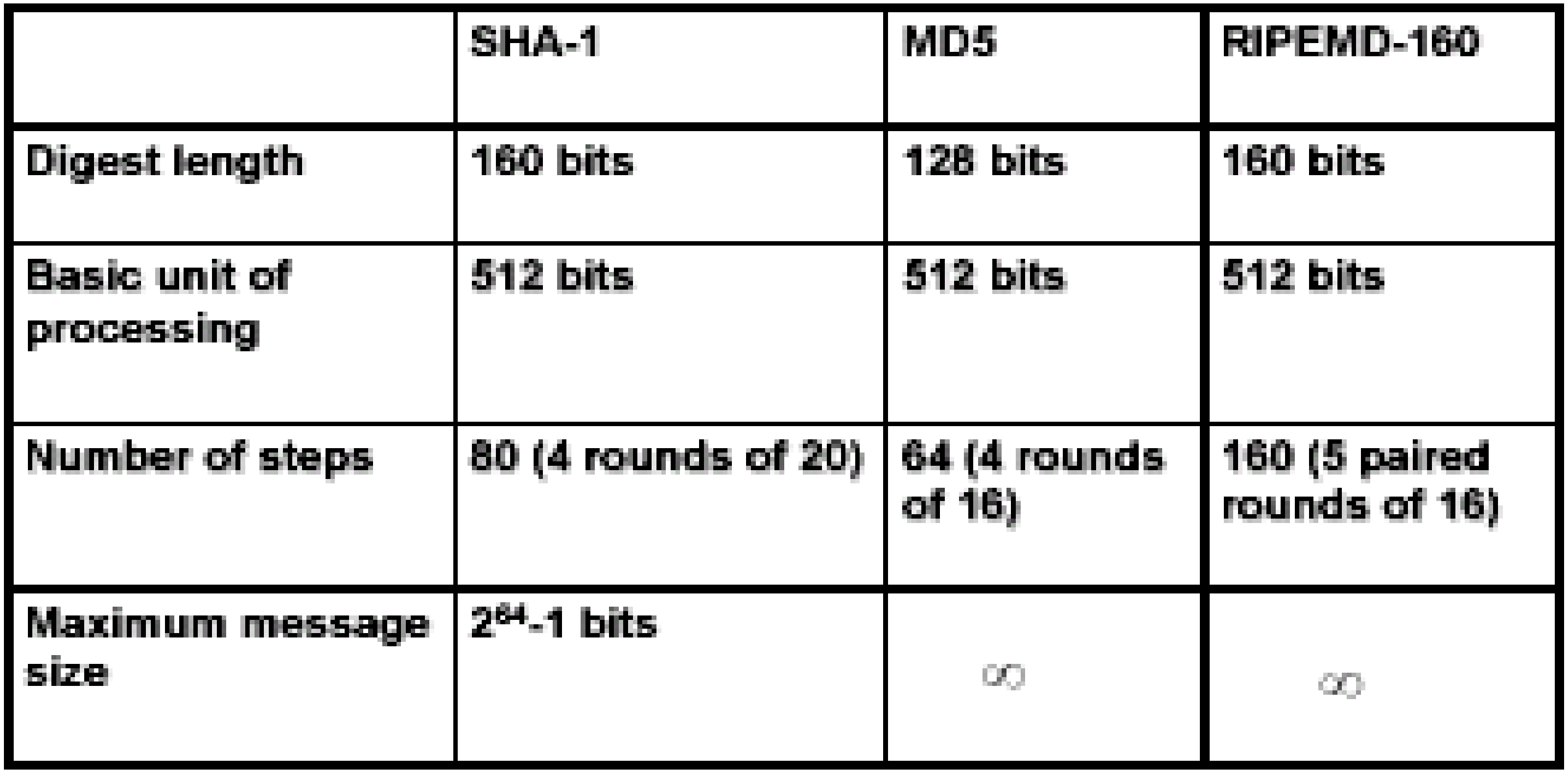
RIPEMD-160 Strength

- Resistance to brute-force attack
  - All 3 algorithms are invulnerable to attacks against weak collision resistance
  - MD5 is highly vulnerable to birthday attack on strong collision resistance
  - SHA-1 and RIPEMD-160 are safe for the foreseeable future
- Resistance to cryptanalysis
  - Designed specifically to resist known cryptanalytic attacks
  - The use of two lines of processing
    - gives RIPEMD-160 added complexity
    - should make cryptanalysis more difficult than SHA-1

Speed

- Speed
  - All 3 algorithms rely on addition modulo 2^{32} and simple bitwise logical operations
  - The added complexity and number of steps of SHA-1 and RIPEMD-160 does lead to slowdown compared to MD5

Comparison



	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	∞	∞

Performance Comparison

Name	Bit-length	Rounds x Steps per Round	Relative Speed
MD4	128	3 x 16	1.00
MD5	128	4 x 16	0.68
RIPEMD-128	128	4 x 16 twice (in parallel)	0.39
SHA-1	160	4 x 20	0.28
RIPEMD-160	160	5 x 16 twice (in parallel)	0.24

HMAC

- Developing a MAC derived from a cryptographic hash code
- Motivations
  - generally execute faster in software than symmetric block ciphers
  - No export restrictions from US or other countries for cryptographic hash code

HMAC (cont'd)

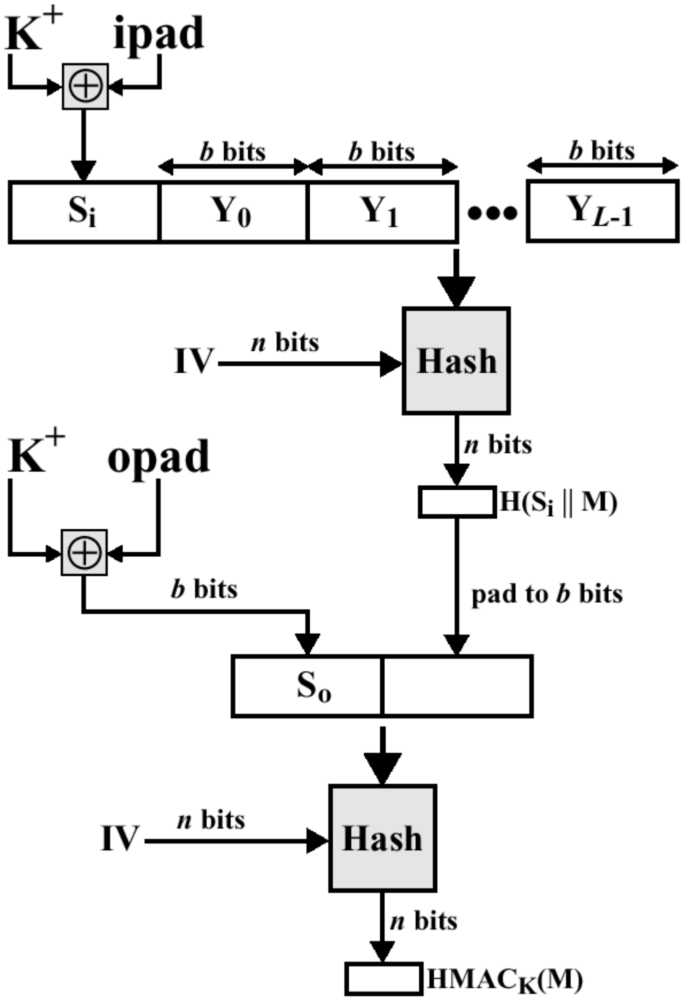
- HMAC Design Objectives [RFC2104]
  - To use available hash functions.
  - To allow for easy replaceability of the embedded hash function
  - To preserve the original performance
  - To use and handle keys in simple way
  - To have a well understood cryptographic analysis of the strength of the authentication mechanism

HMAC Algorithm

$$HMAC_K = H[(K^+ \oplus opad) \| H[K^+ \oplus ipad \| M]]$$

1. Append zeros to the left end of K to create a *b*-bit string K^+
2. XOR K^+ with ipad to produce the *b*-bit block S_i
3. Append M to S_i
4. Apply H to the stream generated in step 3

Algorithm (cont'd)



5. XOR K^+ with opad to produce the *b*-bit block S_o
6. Append the hash result from step 4 to S_o
7. Apply H to the stream generated in step 6 and output the result

Algorithm Logic

- Pseudorandom generation of 2 keys from K
  - XOR with ipad/opad results in flipping one-half of the bits of K -> S_i/S_o
- More efficient implementation is possible.

Security of HMAC

- Depends on the cryptographic strength of the underlying hash function
- Generally expressed in terms of prob. of successful forgery with a given amount of time and number of message-MAC pairs