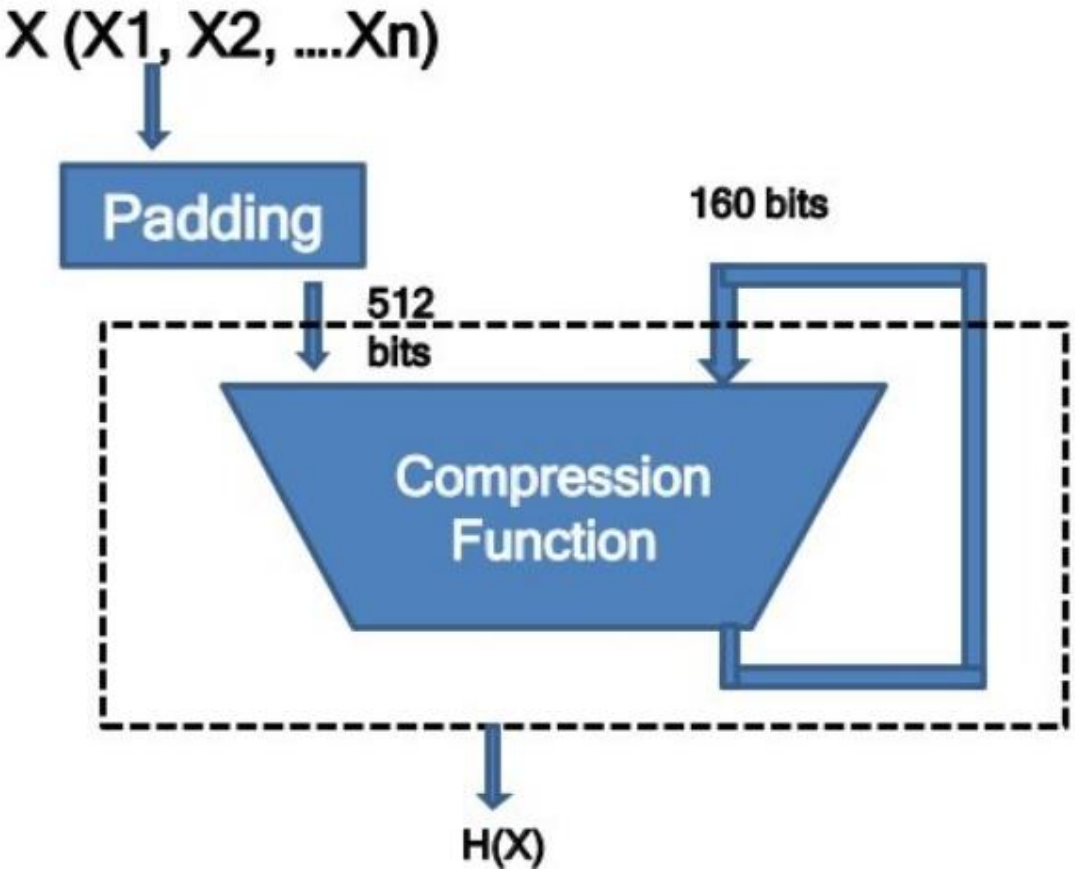
SHA - 1

- SHA1: Secure Hash Algorithm 1
- Designed by the United States National Security Agency
- Produces hash value known as Message Digest
- Works for any input message that is less than 2^{64} bits
- produces160 bits length message digest
- Infeasible to retain the original message from the message digest
- Same message digest to be produced from both sender and receiver
- Purpose: Authentication , not Encryption
- widely used in security applications and protocols, including TLS, SSL, PGP, SSH, IPSec and S/MIME

SHA-1 Steps



1. Append padding bits
2. Append Length
3. Initialize hash bufffer
4. SHA processing
5. Output

SHA-1 Steps

Step 1: Append Padding bits

padding bits are added to the original message to make the original message equal to a value divisible by 512.

Example –

- The massage padding is applied to the last data block such that SHA-1 can process the data of n×512 bits.
- The last two words (64 bits) of padded message are reserved of the original message length (in bits).
- Input message – ‘abcde’ – 40 bits
- 01100001 01100010 01100011 01100100 01100101.
- After ‘1” is appended, 407 ‘0’ are required to complete 448 bits. In Hex, this can be written as:

61626364 65800000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000

SHA-1 Steps

Step 2: Append length

The rest two words are preserved for the original message

length.

As per example, length of msg = 40 = “00000000 00000028” (Hexadecimal Value).

As a result, the passed massage is

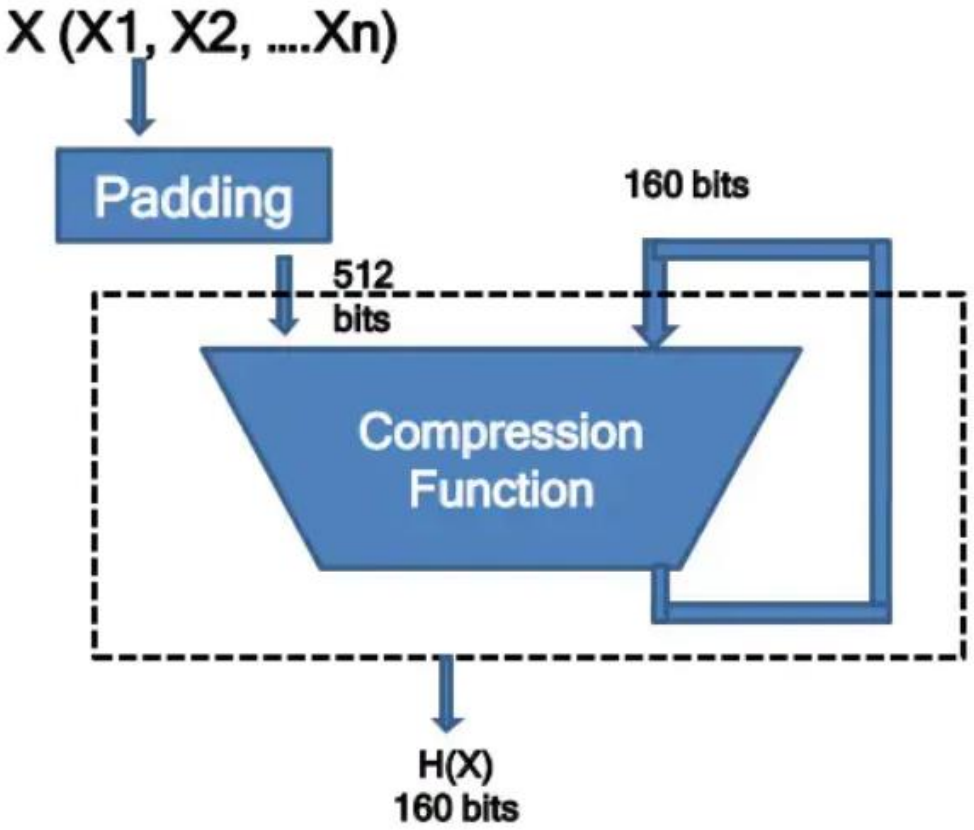
61626364 65800000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000028.

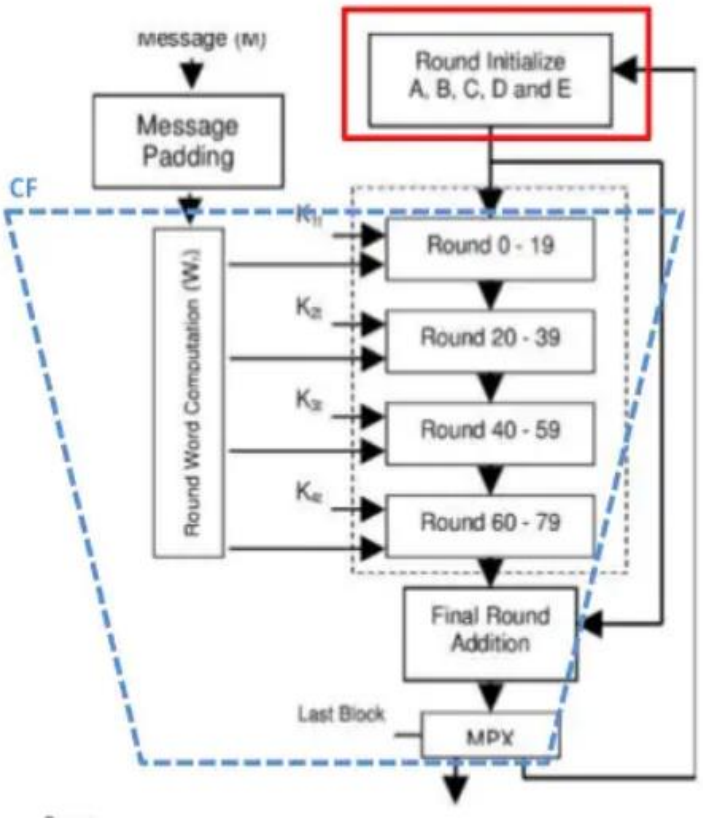
SHA-1 Steps



- This 512 bits input to the compression function
- The message divided into 16 words.
- Each word consists of 32 bits.
- 512/32 =16 words

SHA-1 Steps

Step 3: Initialize the hash buffer



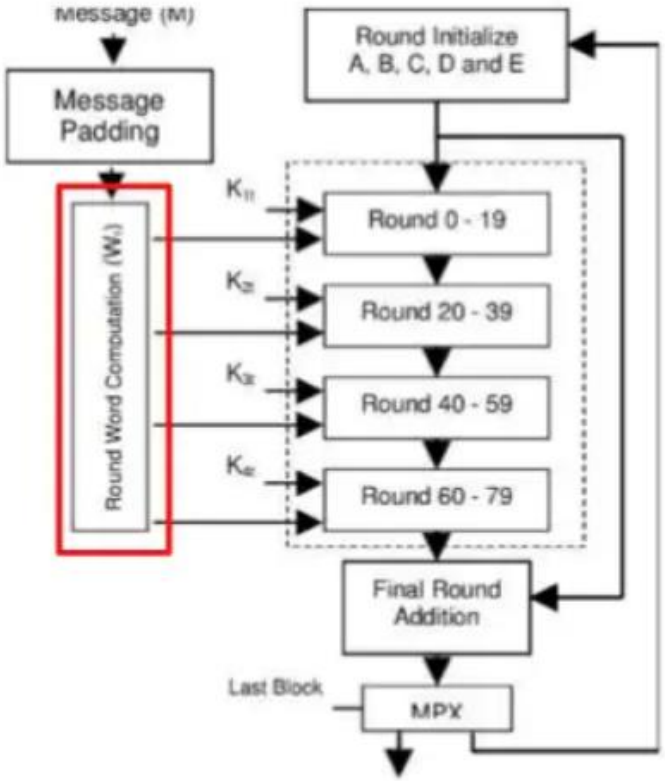
Initial values of Ho are predefined and stored in registers ABCDE

H	Hex values
Ho(A)	01234567
Ho(B)	89ABCDEF
Ho(C)	FEDCBA98
Ho(D)	76543210
Ho(E)	C3D2E1F0

These initial values are used in Round 0.

SHA-1 Steps

Step 4: SHA Processing



Word assigning to rounds:

SHA1 has 80 rounds defined.

The Message Scheduler Algorithm schedules each word to rounds as:

- W_0 → Round 0
- W_1 → Round 1

  .....
- W_{15} → Round 15
- W_{16} → Round 16

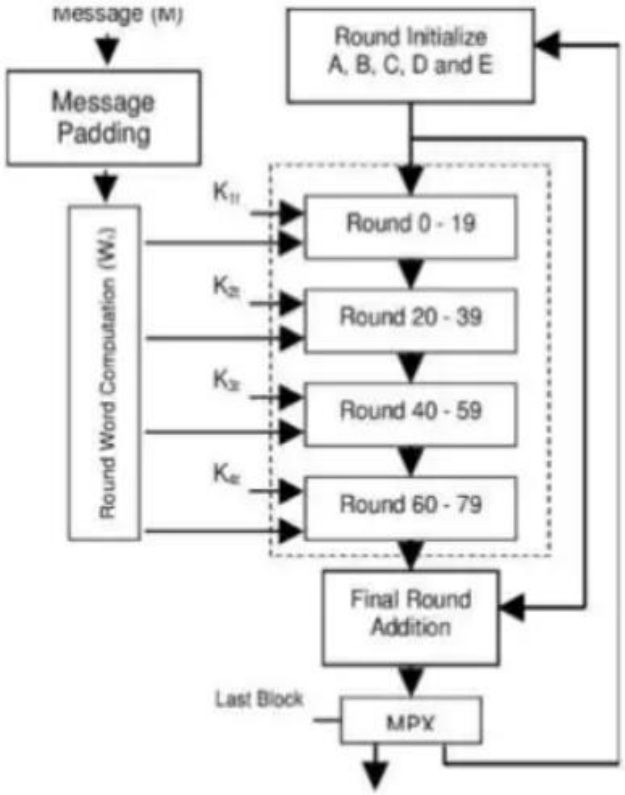
  .....
- W_{79} → Round 79

W_t is calculated

Each round = 20 iterations. Total iterations = 80

SHA-1 Steps

Step 4: SHA Processing



Word assigning to other rounds:

For others (i.e round 16- 79)

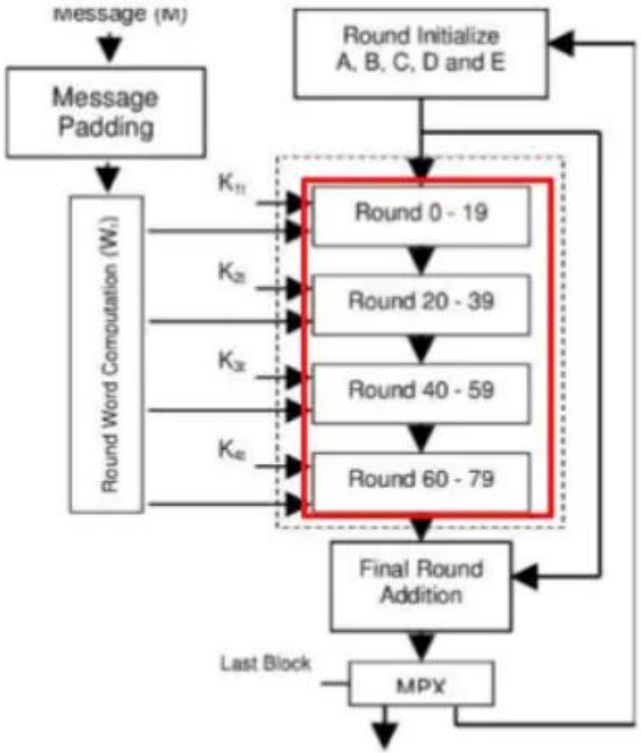
$W_{[t]} = S^1(w_{[t-16]} \text{ XOR } W_{[t-14]} \text{ XOR } W_{[t-8]} \text{ XOR } W_{[t-3]})$

For example: when round is 16,

- $W_{[16]} = S^1(w_{[16-16]} \text{ XOR } W_{[16-14]} \text{ XOR } W_{[16-8]} \text{ XOR } W_{[16-3]})$
- Here W_0, W_2, W_8 and W_{13} are XORed.
- The output is the new word for round 16.

SHA-1 Steps

Step 4: SHA Processing



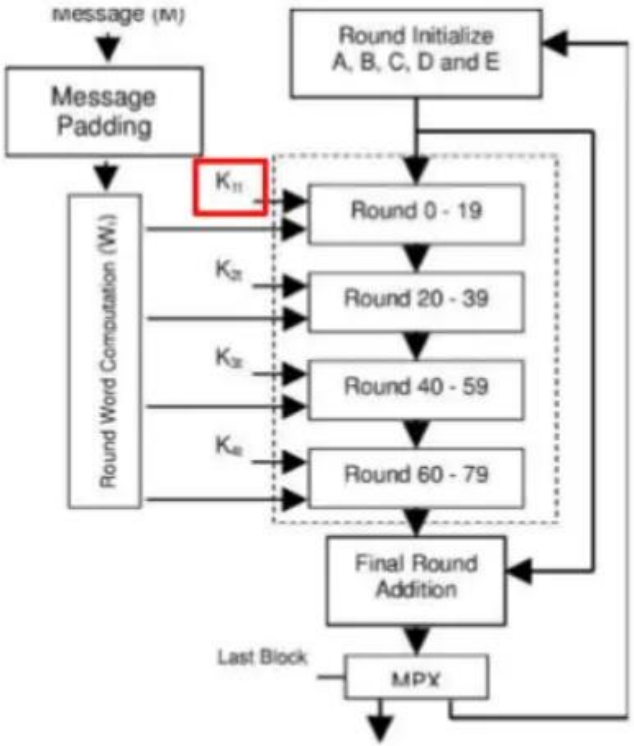
Division of stages:

Stages	Round
t=1	0 to 19
t=2	20 to 39
t=3	40 to 59
t=4	60 to 79

Each stage has 20 rounds.

SHA-1 Steps

Step 4: SHA Processing



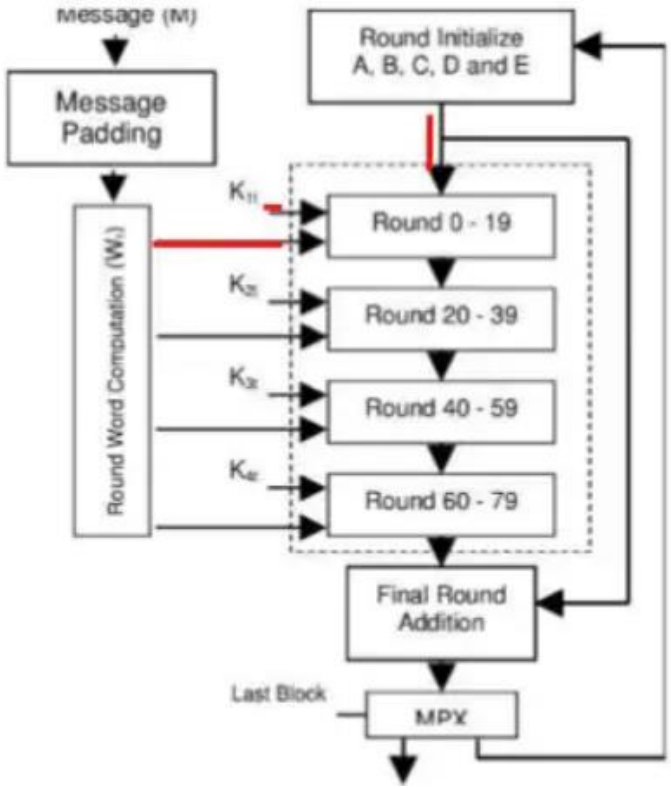
Constant values:

At each stage:

Stage	Predefined value of k
t_1	K_1=0X5A827999
t_2	K_2= 0X6ED9EBA1
t_3	K_3=0X8F1BBCDC
t_4	K_4=0XCA62C1D6

SHA-1 Steps

Step 4: SHA Processing



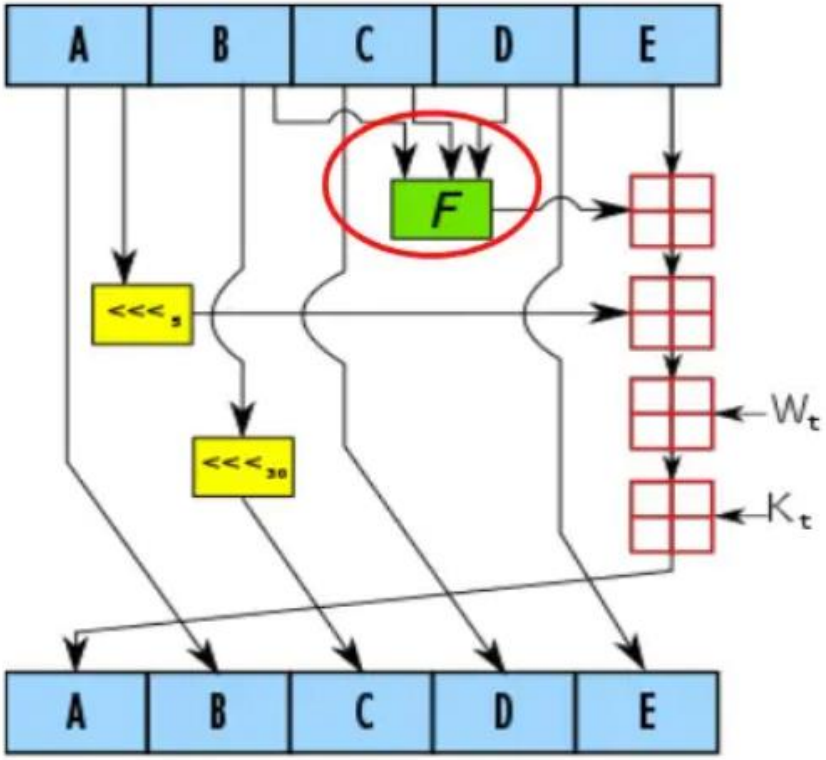
Process in each round:

Each round takes 3 inputs:

- 32 bit word form 512 bit block (i.e.W_t)
- The values from register ABCDE
- Constant K_t

SHA-1 Steps

Step 4: SHA Processing

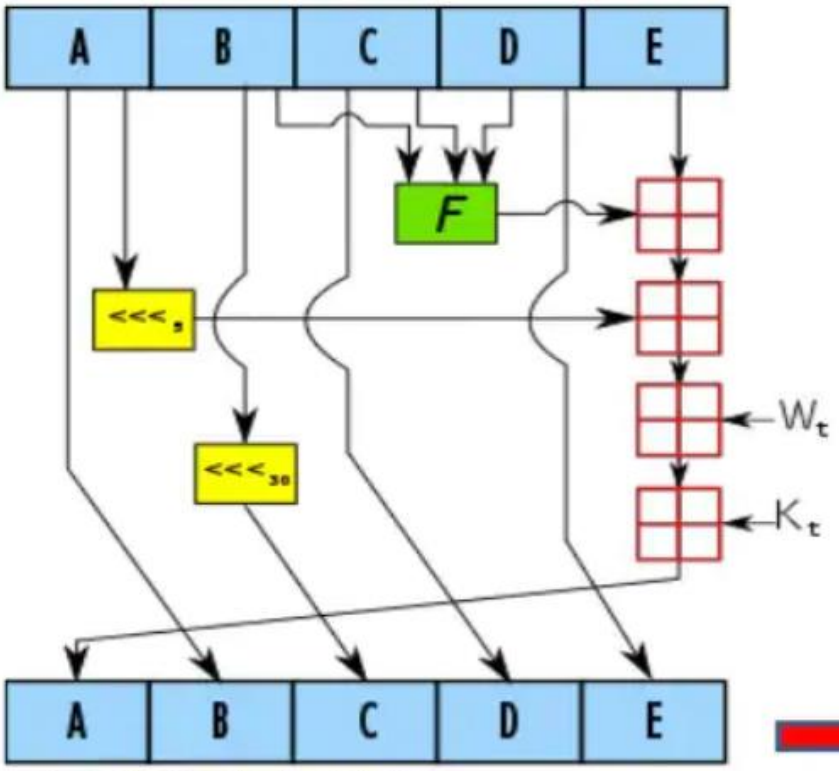


F_t at different stages:

Stage	F_t
t_1	$F_t(B.C,D)$ = (B AND C) OR ((NOT B) AND D)
t_2	$F_t(B.C,D)$ = B XOR C XOR D
t_3	$F_t(B.C,D)$ = (B AND C) OR (B AND D) OR (C AND D)
t_4	$F_t(B.C,D)$ = B XOR C XOR D

SHA-1 Steps

Step 4: SHA Processing



Source: https://en.wikipediaorg/wiki/SHA-1

Denotes addition module 2^{32}

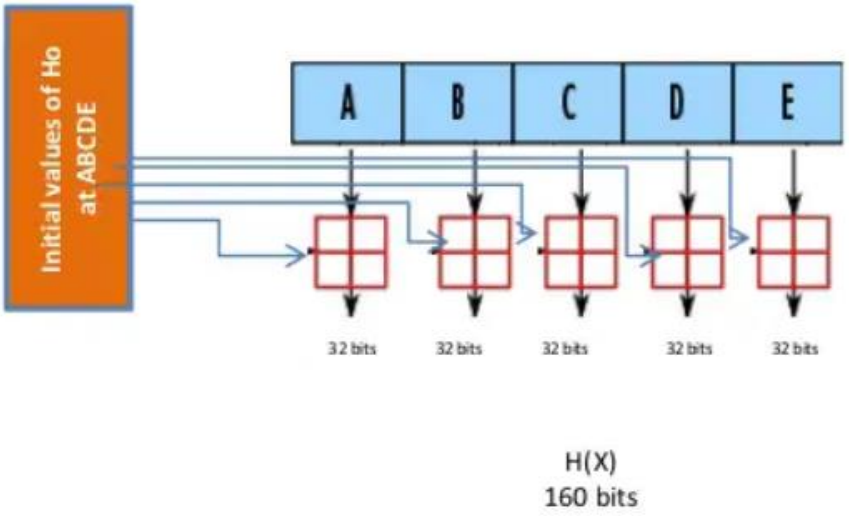
At each Round:

- Output of F_t and E are added
- Value in register A is 5 bit circular-left shifted.
- This then added to previous sum.
- W_t is added
- K_t introduced
- B is circular-left shifted by 30 bits.

New values for next round

SHA-1 Steps

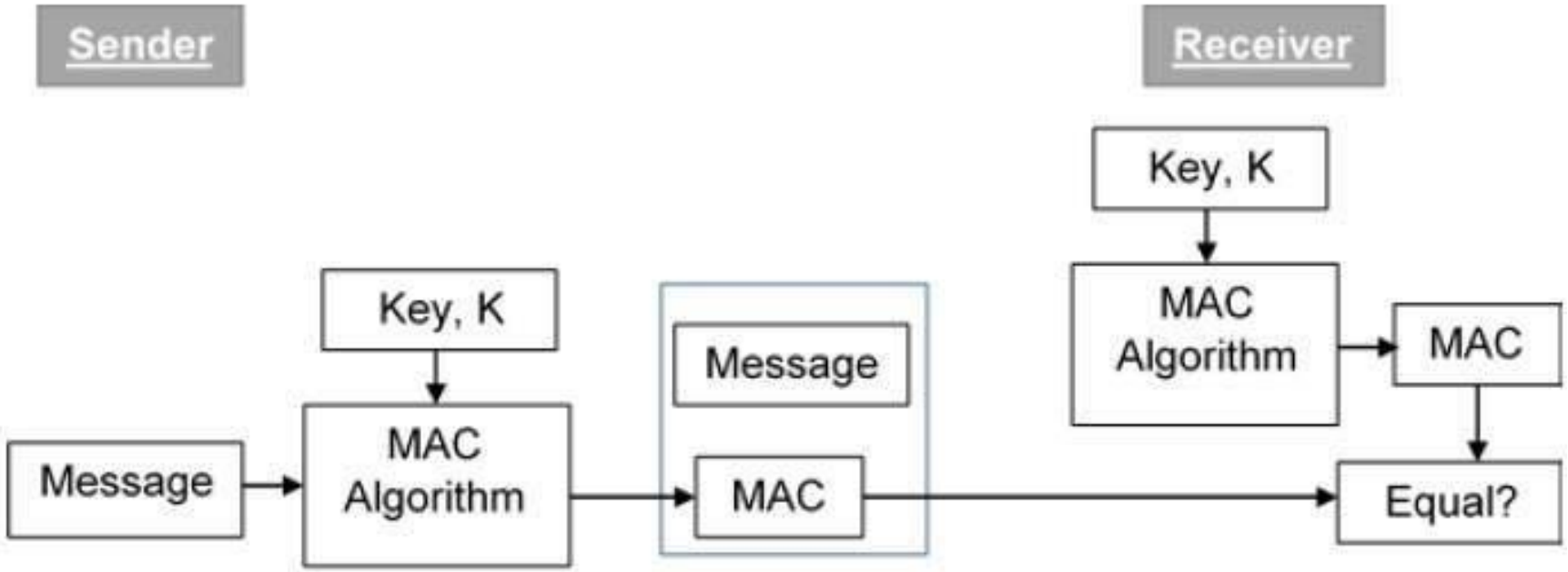
Step 4: The Output



After Final Round:

- The 160 bit output from the final round is modulo added to the initial predefined values of Ho at registers ABCDE.
- Output obtained thus is a 160 bit hash code.

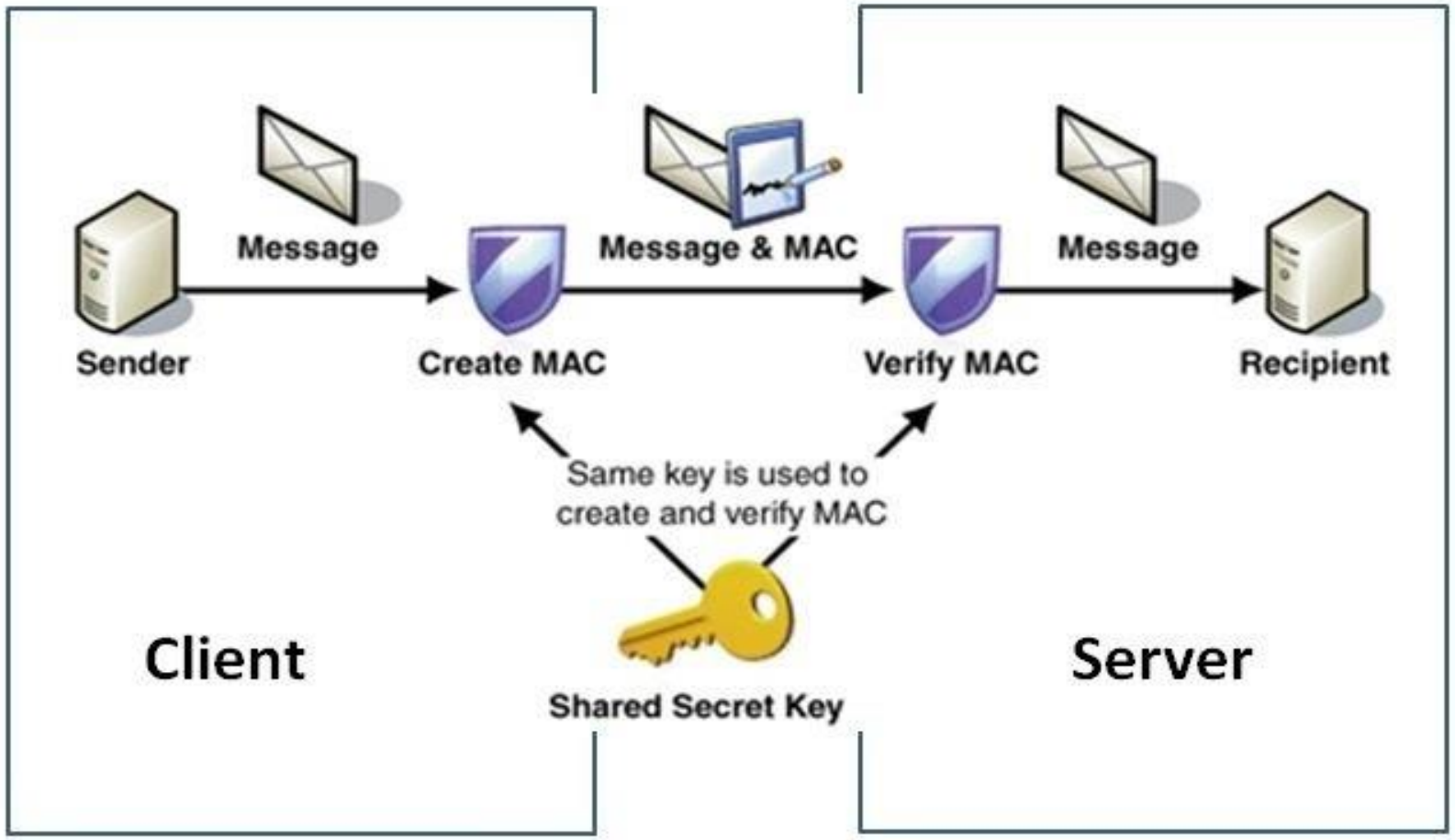
Let’s recall MAC ..



Hash based Message Authentication Code (HMAC)

- Hash-based message authentication code (HMAC) is a mechanism for calculating a message authentication code involving a hash function in combination with a secret key. This can be used to verify the integrity and authenticity of a message.
- HMACs are almost similar to digital signatures. They both enforce integrity and authenticity. They both use cryptography keys. And they both employ hash functions.
- The main difference is that digital signatures use asymmetric keys, while HMACs use symmetric keys (no public key).

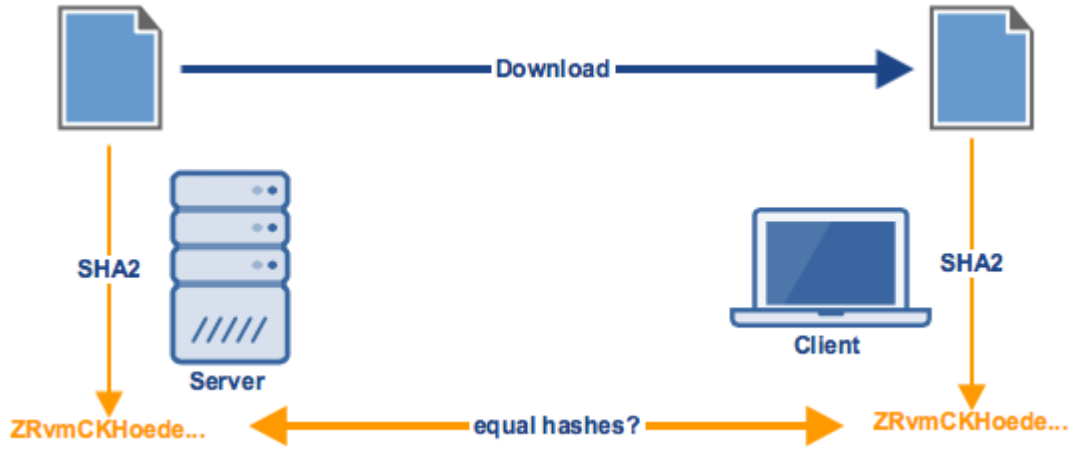
HMAC Authentication



How does HMAC Work ?

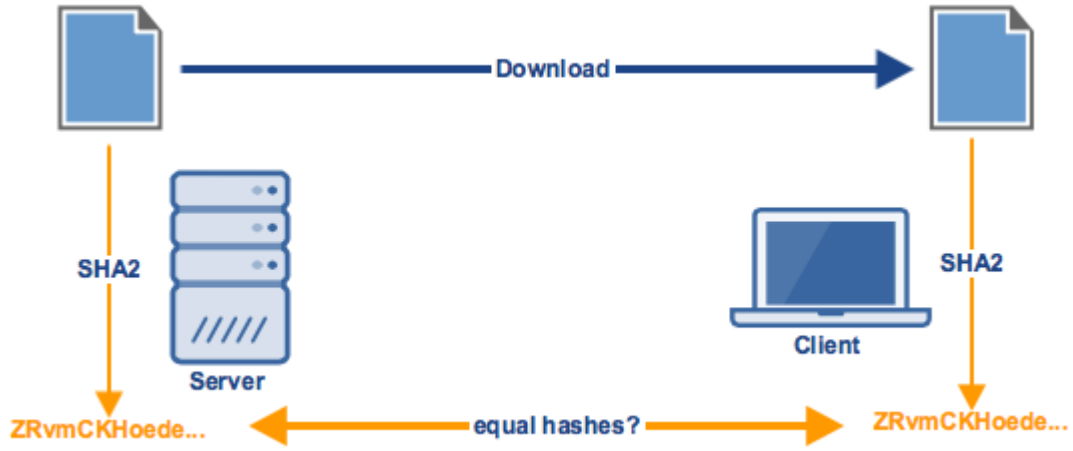
A data integrity check on a file transfer.

- Let's say a client application downloads a file from a remote server. It's assumed that the client and server have already agreed on a common hash function, for example SHA2.



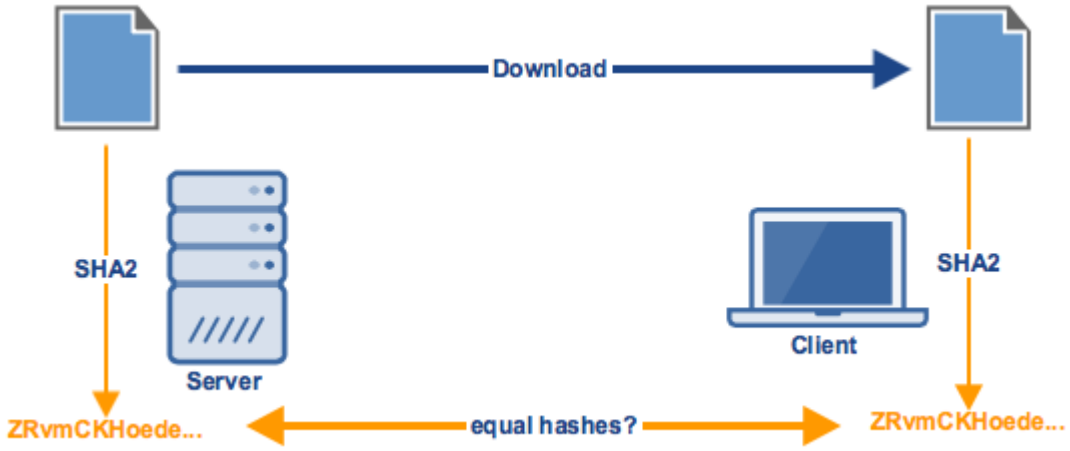
How does HMAC Work ?

- Before the server sends out the file, it first obtains a hash of that file using the SHA2 hash function. It then sends that hash (ex. a message digest) along with the file itself.
- Upon receiving the two items (ex. the downloaded file and the hash), the client obtains the SHA2 hash of the downloaded file and then compares it with the downloaded hash.
- If the two match, then that would mean the file was not tampered with.



How does HMAC Work ?

- If an attacker manages to intercept the downloaded file, alter the file's contents, and then forward the altered file to the recipient, that malicious act won't go unnoticed.
- That's because, once the client runs the tampered file through the agreed hash algorithm, the resulting hash won't match the downloaded hash.
- This will let the receiver know the file was tampered with during transmission.



How does HMAC Work ?

Authenticity Check

- An HMAC employs both a hash function and a shared secret key.
- A shared secret key provides exchanging parties a way to establish the authenticity of the message.
- That is, it provides the two parties a way of verifying whether both the message and MAC (more specifically, an HMAC) they receive really came from the party they're supposed to be transacting with.

How does HMAC Work ?

Suitable for File Transfers

Efficiency - hash functions can take a message of arbitrary length and transform it into a fixed-length digest. That means, even if you have relatively long messages, their corresponding message digests can remain short, allowing you to maximize bandwidth.

HMAC Structure

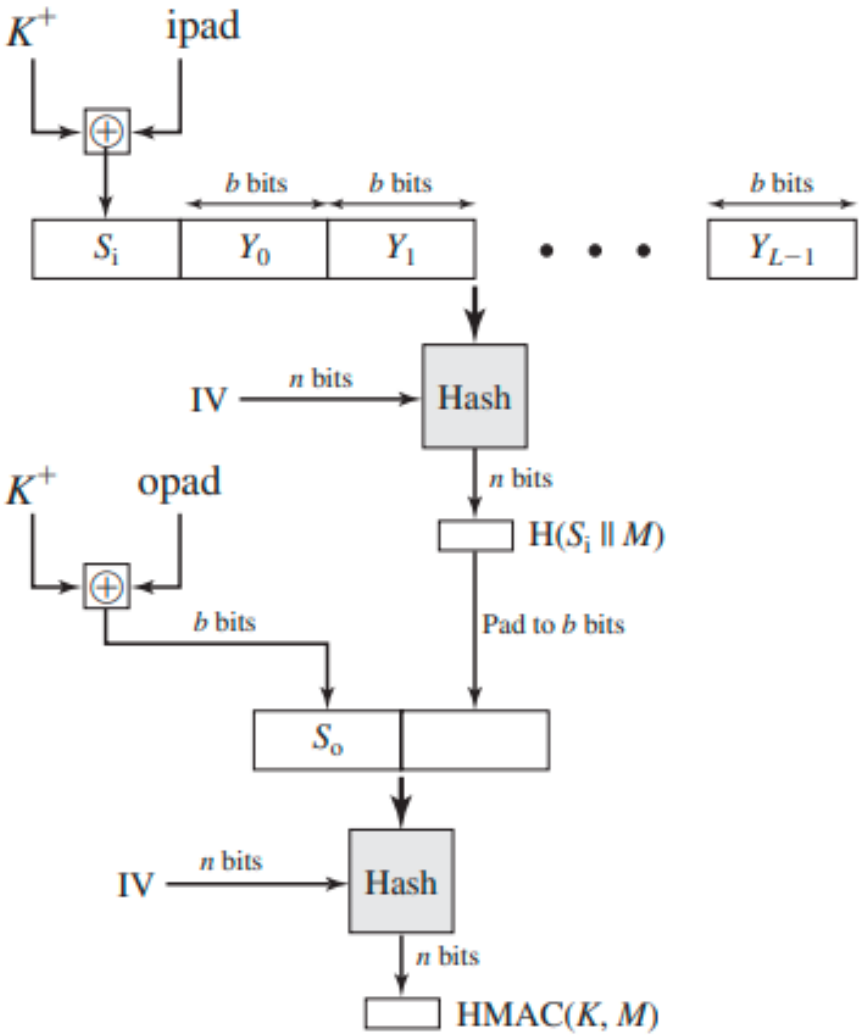


Figure 21.4 **HMAC Structure**

HMAC Structure

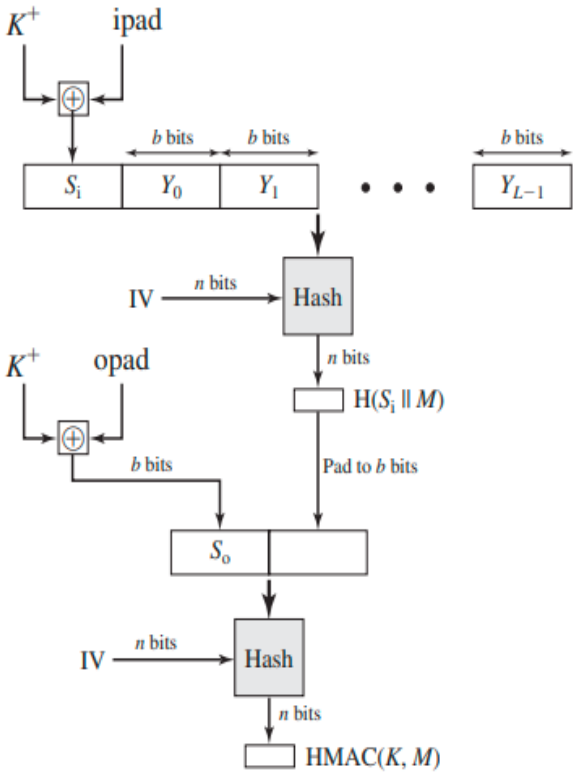


Figure 21.4 **HMAC Structure**

Figure 21.4 illustrates the overall operation of HMAC. Define the following terms:

- II = embedded hash function (e.g., SHA)
- M = message input to HMAC (including the padding specified in the embedded hash function)
- Y_i = ith block of $M, 0 \leq i \leq (L - 1)$
- L = number of blocks in M
- b = number of bits in a block
- n = length of hash code produced by embedded hash function
- K = secret key; if key length is greater than b, the key is input to the hash function to produce an n-bit key; recommended length is $\geq n$
- K^+ = K padded with zeros on the left so that the result is b bits in length
- ipad = 00110110 (36 in hexadecimal) repeated $b/8$ times
- opad = 01011100 (5C in hexadecimal) repeated $b/8$ times

Then HMAC can be expressed as follows:

$$\text{HMAC}(K, M) = \text{H}[(K^+ \oplus \text{opad}) \| \text{H}[K^+ \oplus \text{ipad}] \| M]]$$

In words,

1. Append zeros to the left end of K to create a b-bit string K^+ (e.g., if K is of length 160 bits and $b = 512$, then K will be appended with 44 zero bytes 0x00).
2. XOR (bitwise exclusive-OR) 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.
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.

Reference - Stallings

MAC and HMAC reference

- Stalling