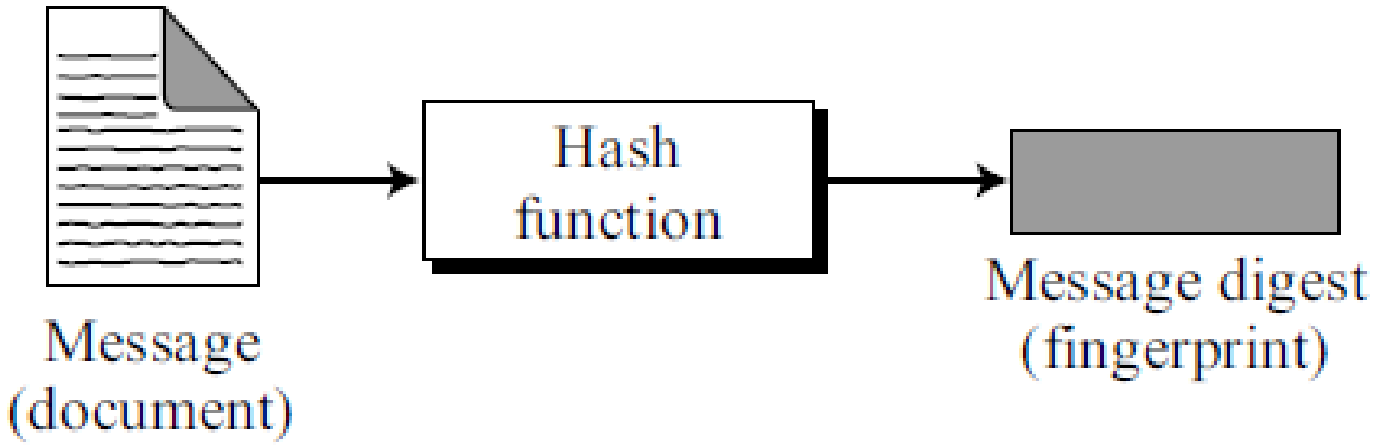
Hashing

Message Integrity And Message Authentication

- One way to preserve the integrity of a document is through the use of a fingerprint
- If Alice needs to be sure that the contents of her document will not be changed, she can put her fingerprint at the bottom of the document.
- Eve cannot modify the contents of this document or create a false document because she cannot forge Alice's fingerprint.
- To ensure that the document has not been changed, Alice's fingerprint on the document can be compared to Alice's fingerprint on file.
- If they are not the same, the document is not from Alice

Message and Message Digest

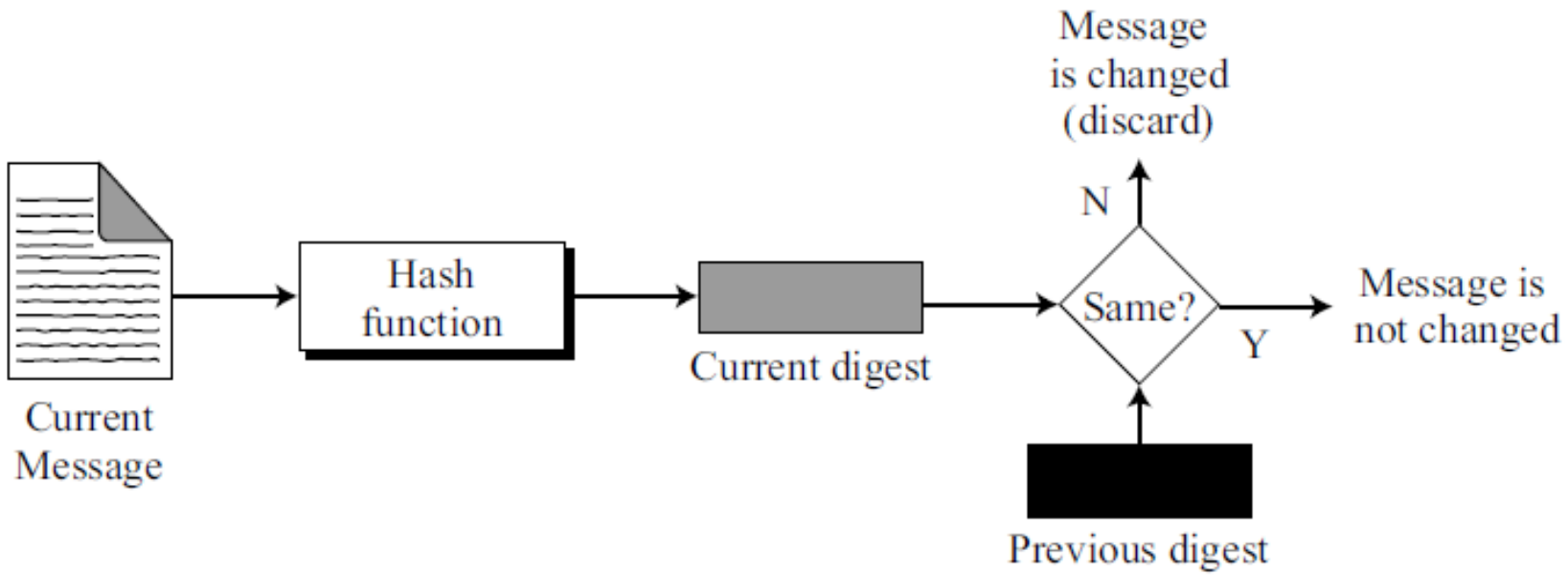
- The electronic equivalent of the document and fingerprint pair is the message and digest pair.
- To preserve the integrity of a message, the message is passed through an algorithm called a cryptographic hash function.
- The function creates a compressed image of the message that can be used like a fingerprint



- The two pairs (document/fingerprint) and (message/message digest) are similar, with some differences.
- The document and fingerprint are physically linked together.
- The message and message digest can be unlinked (or sent) separately, and, most importantly, the message digest needs to be safe from change.

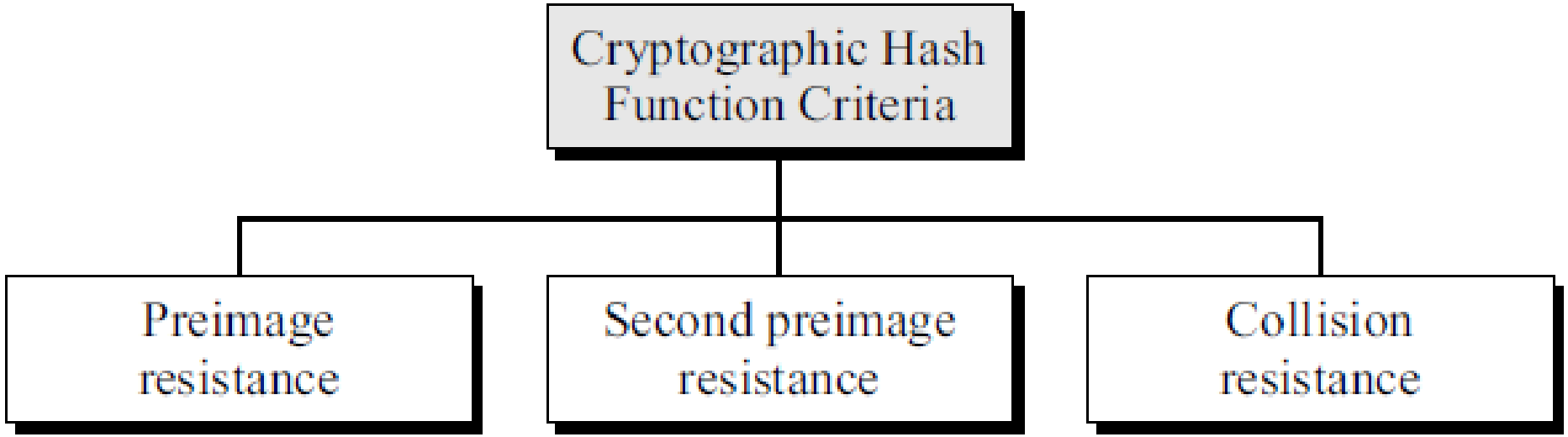
Checking Integrity

- To check the integrity of a message, or document, we run the cryptographic hash function again and compare the new message digest with the previous one.
- If both are the same, we are sure that the original message has not been changed



Criteria of a cryptographic hash function

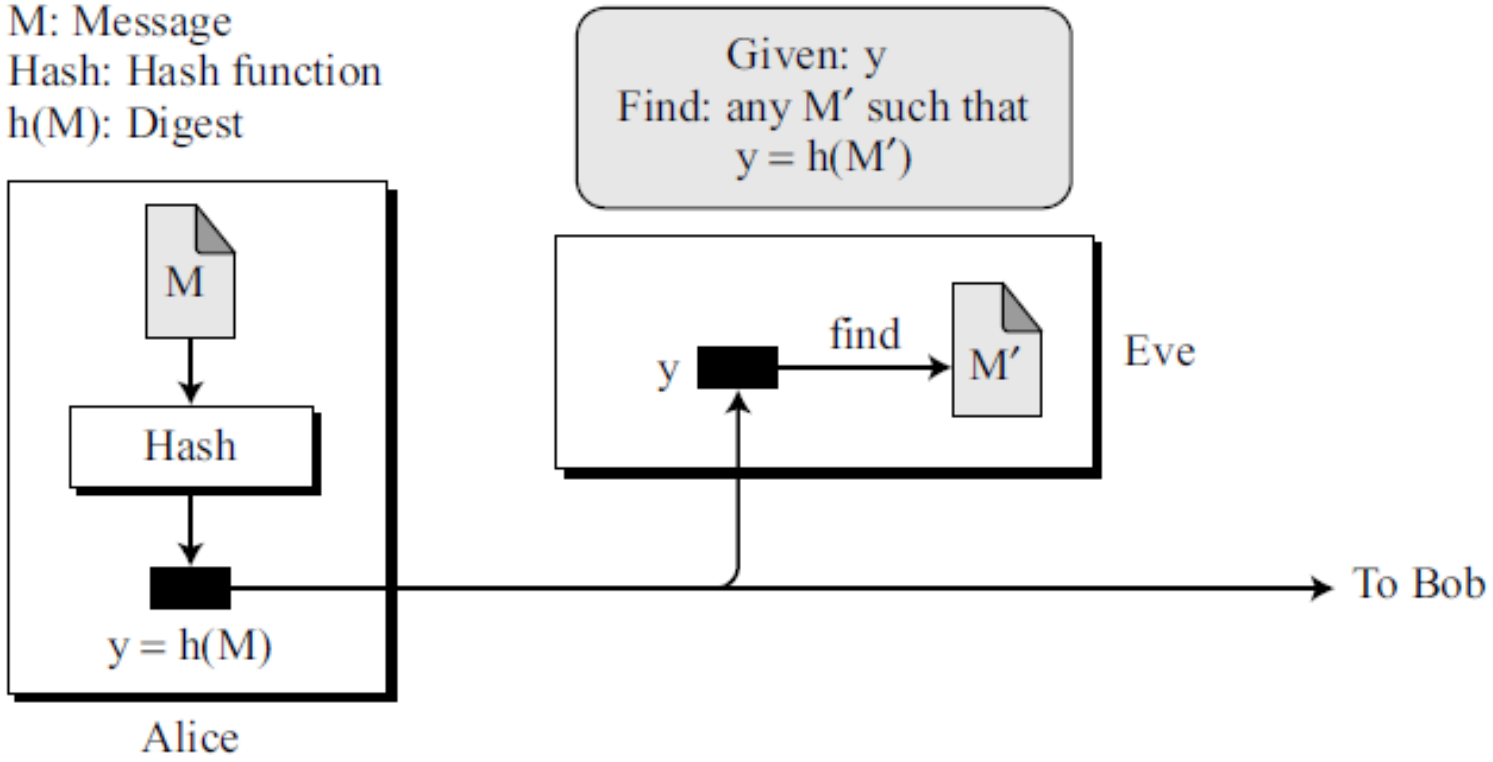
Hash function produces a digest of fixed length for a message of arbitrary length



Preimage Resistance

Hash function produces a digest of fixed length for a message of arbitrary length

- A cryptographic hash function must be preimage resistant.
- Given a hash function h and y = h(M), it must be extremely difficult for Eve to find any message, M′, such that y = h(M′)



- If the hash function is not preimage resistant, Eve can intercept the digest h(M) and create a message M′. Eve can then send M′ to Bob pretending it is M.

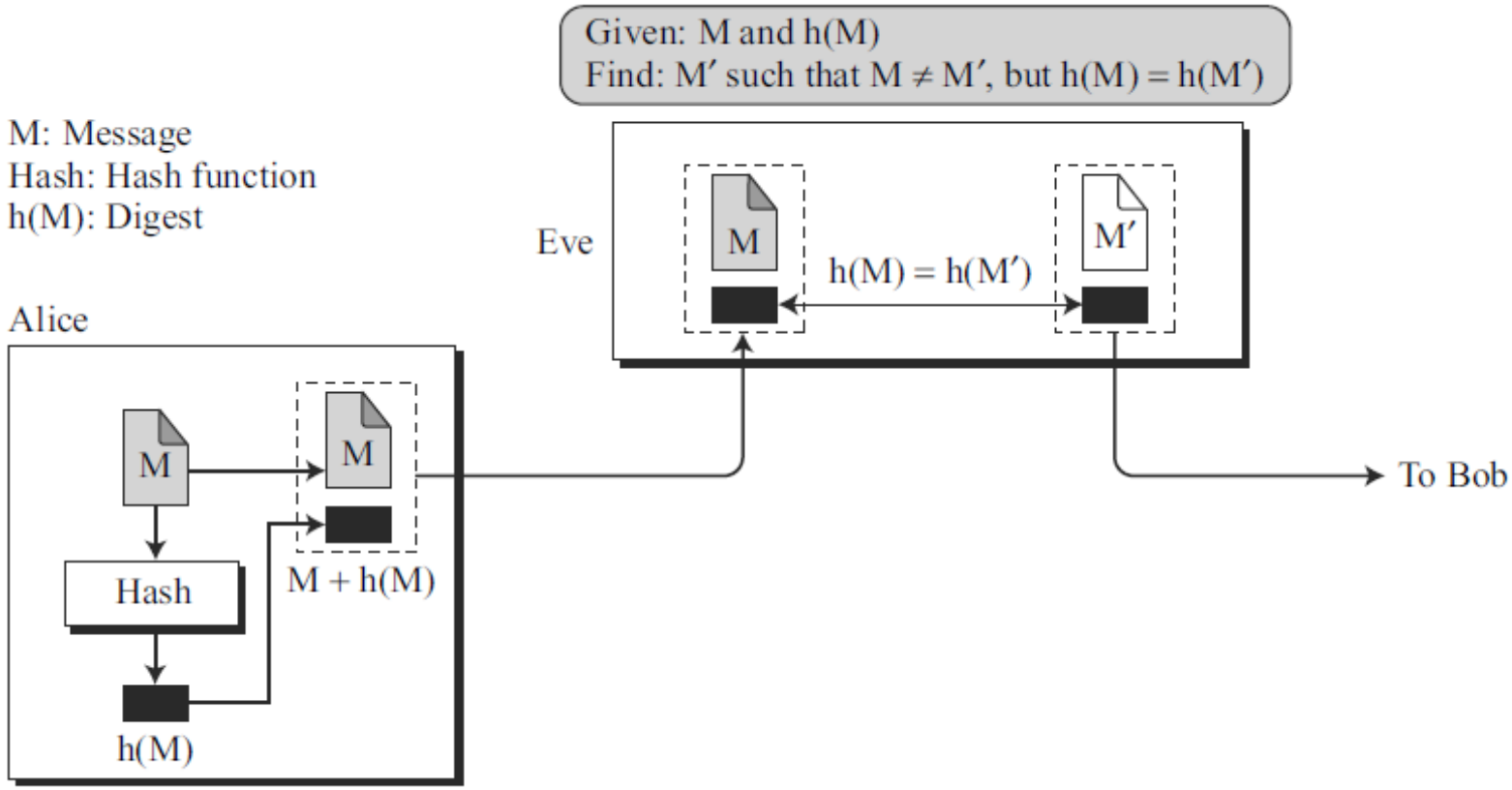
Second Preimage Resistance

- ensures that a message cannot easily be forged
- If Alice creates a message and a digest and sends both to Bob, this criterion ensures that Eve cannot easily create another message that hashes to the exact same digest
- In other words, given a specific message and its digest, it is impossible

(or at least very difficult) to create another message with the same digest

- Eve intercepts (has access to) a message M and its digest h(M). She creates another message M′≠ M, but h(M) = h(M′)
- Eve sends the M′ and h(M′) to Bob. Eve has forged the message.

Second Preimage Resistance

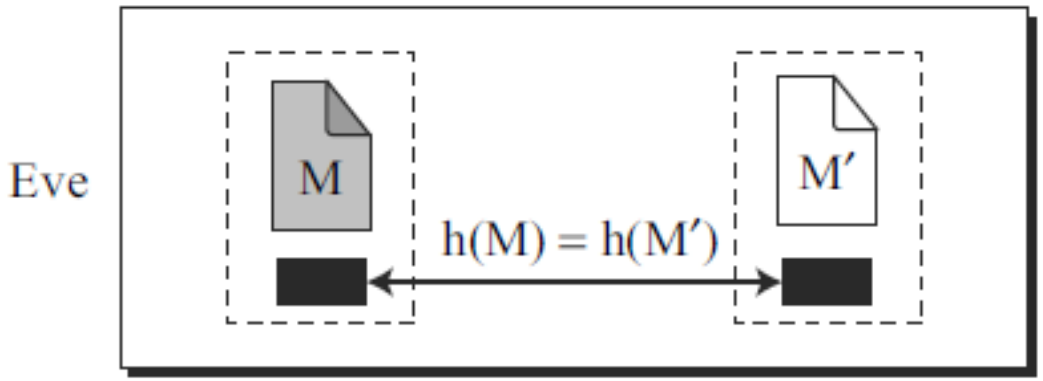


Collision Resistance

- ensures that Eve cannot find two messages that hash to the same digest.
- Here the adversary can create two messages (out of scratch) and hashed to the same digest

M: Message
Hash: Hash function
h(M): Digest

Find: M and M′ such that $M \neq M'$, but $h(M) = h(M')$



Serial and Parallel Hashing

- Arbitrarily long messages can be compressed using a fixed input size hash function by applying 2 methods
- Serial and parallel
- Serial: Message ***m*** is first split into blocks of size ***n*** so that

  m=(m1, m2, …..mk) for i=1,2, …k.
- If the last block is shorter than ***n*** bits, it is padded with zeros to full length

h1= h(m1, m2) , h2= h(m3, h1)….

hi= h(m_{i+1}, h_{i-1}) , d=h(m_k, h_{k-2})

Result d is the hash of the whole message m

Parallel Hashing

- Split the message ***m*** into ***ℓ*** blocks of size ***n***

$$m=(m1, m2, \ldots..m_{\ell})$$

Last block is padded to the full length if necessary

Assume that the number of blocks is $2^{k-1} < \ell <= 2^{k}$

$2^{k} - \ell$ blocks all with zero bits are appended to the message ***m***

Resulting message is

$$m=(m1, m2, \ldots..m_{\ell} \ldots\ldots m_{2^{k}})$$

$$h_i^{1} = h(m_{2i-1}, m_{2i}) \quad for\ i= 1...., 2^{k-1}$$

$$h_i^{j} = h(h_{2i-1}^{j-1}, h_{2i}^{j-1}) \quad for\ i= 1...., 2^{k-l} \quad and\ j= 2, \ldots k-1$$

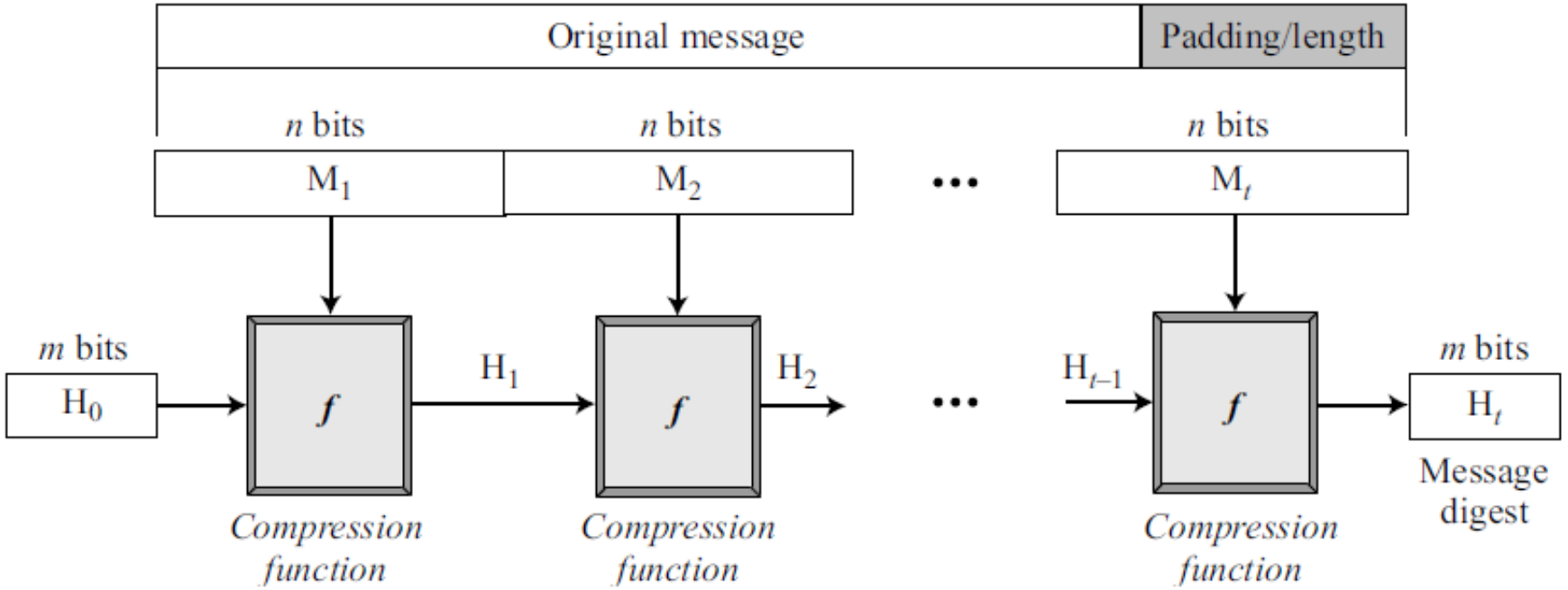
$$D(m) = h(h_1^{k-1}, h_2^{k-1})$$

Iterated Hash Function

- All cryptographic hash functions need to create a fixed-size digest out of a variable-size message
- Creating such a function is best accomplished using iteration
- Instead of using a hash function with variable-size input, a function with fixed-size input is created and is used a necessary number of times
- The fixed-size input function is referred to as a compression function
- It compresses an n-bit string to create an m-bit string where n is normally greater than m. The scheme is referred to as an iterated cryptographic hash function.

Merkle-Damgard Scheme

- The Merkle-Damgard scheme is an iterated hash function that is collision resistant if the compression function is collision resistant



The scheme uses the following steps:

1. The message length and padding are appended to the message to create an augmented message that can be evenly divided into blocks of n bits, where n is the size of the block to be processed by the compression function.

2. The message is then considered as t blocks, each of n bits.

We call each block M1, M2,…, Mt.

We call the digest created at t iterations H1, H2,…, Ht.

3. Before starting the iteration, the digest H0 is set to a fixed value, normally called IV (initial value or initial vector).

4. The compression function at each iteration operates on Hi−1 and Mi to create a new Hi.

In other words, we have Hi = ƒ(Hi−1, Mi), where ƒ is the compression function.

5. Ht is the cryptographic hash function of the original message, that is, h(M).

Two Groups of Compression Functions

- The Merkle-Damgard scheme is the basis for many cryptographic hash functions today
- design a compression function that is collision resistant and insert it in the Merkle-Damgard scheme.
- two different approaches in designing a hash function
- In the first approach, the compression function is made from scratch: it is particularly designed for this purpose.
- In the second approach, a symmetric-key block cipher serves as a compression function

Hash Functions Made from Scratch

- ***Message Digest (MD):*** Several hash algorithms were designed by Ron Rivest, referred to as MD2, MD4, and MD5, where MD stands for Message Digest.
- The last version, MD5, is a strengthened version of MD4 that divides the message into blocks of 512 bits and creates a 128-bit digest.
- It turned out that a message digest of size 128 bits is too small to resist collision attack.
- ***Secure Hash Algorithm (SHA):*** The Secure Hash Algorithm (SHA) is a standard

that was developed by the National Institute of Standards and Technology (NIST) and published as a Federal Information Processing standard (FIP 180).

- referred to as Secure Hash Standard (SHS)
- mostly based on MD5.
- includes SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512.

Other Algorithms

- RACE Integrity Primitives Evaluation Message Digest (RIPMED) has several versions.
- RIPEMD-160 is a hash algorithm with a 160-bit message digest
- RIPEMD-160 uses the same structure as MD5 but uses two parallel lines of execution
- HAVAL is a variable-length hashing algorithm with a message digest of size 128, 160, 192, 224, and 256.
- The block size is 1024 bits.

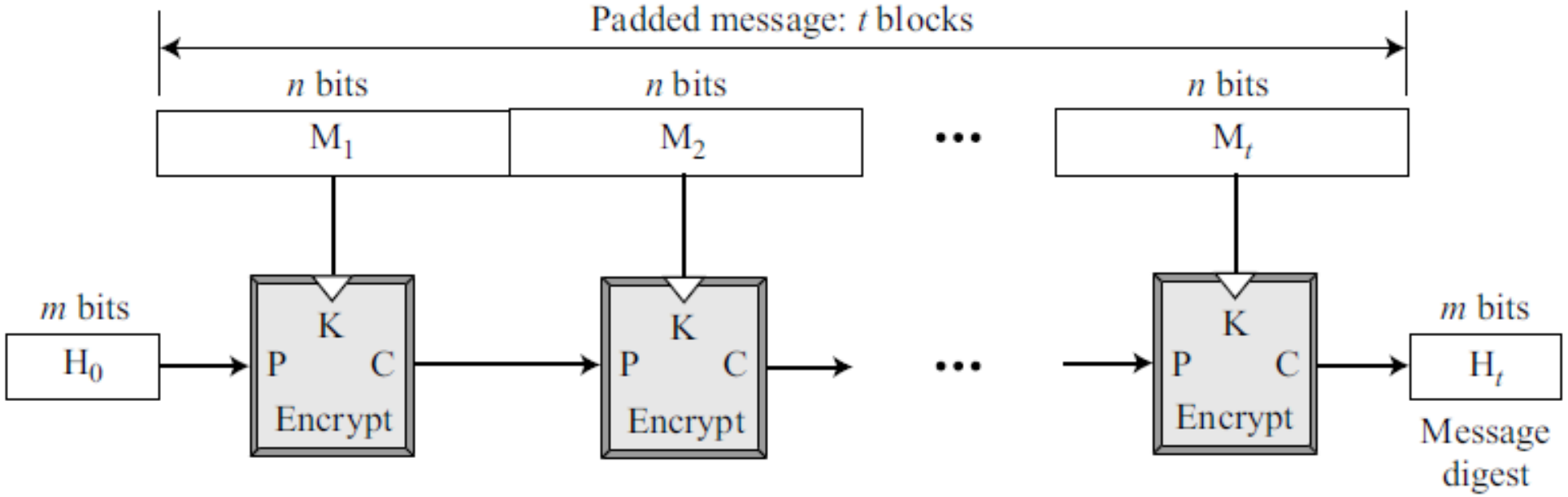
Hash Functions Based on Block Ciphers

- An iterated cryptographic hash function can use a symmetric-key block cipher as a compression function
- The whole idea is that there are several secure symmetric-key block ciphers, such as triple DES or AES, that can be used to make a one-way function instead of creating a new compression function
- The block cipher in this case only performs encryption

Rabin Scheme

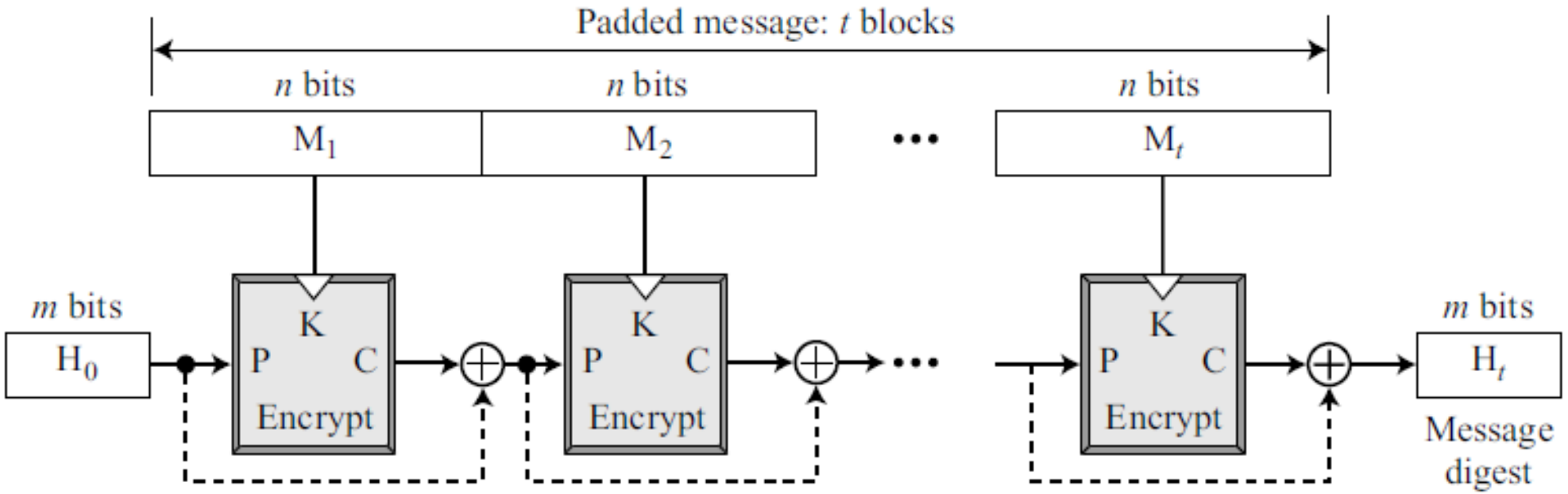
- The iterated hash function proposed by Rabin is very simple
- The Rabin scheme is based on the Merkle-Damgard scheme
- The compression function is replaced by any encrypting cipher
- The message block is used as the key
- The previously created digest is used as the plaintext
- The ciphertext is the new message digest.
- the size of the digest is the size of data block cipher in the underlying cryptosystem.
- For example, if DES is used as the block cipher, the size of the digest is only 64 bits

Rabin Scheme



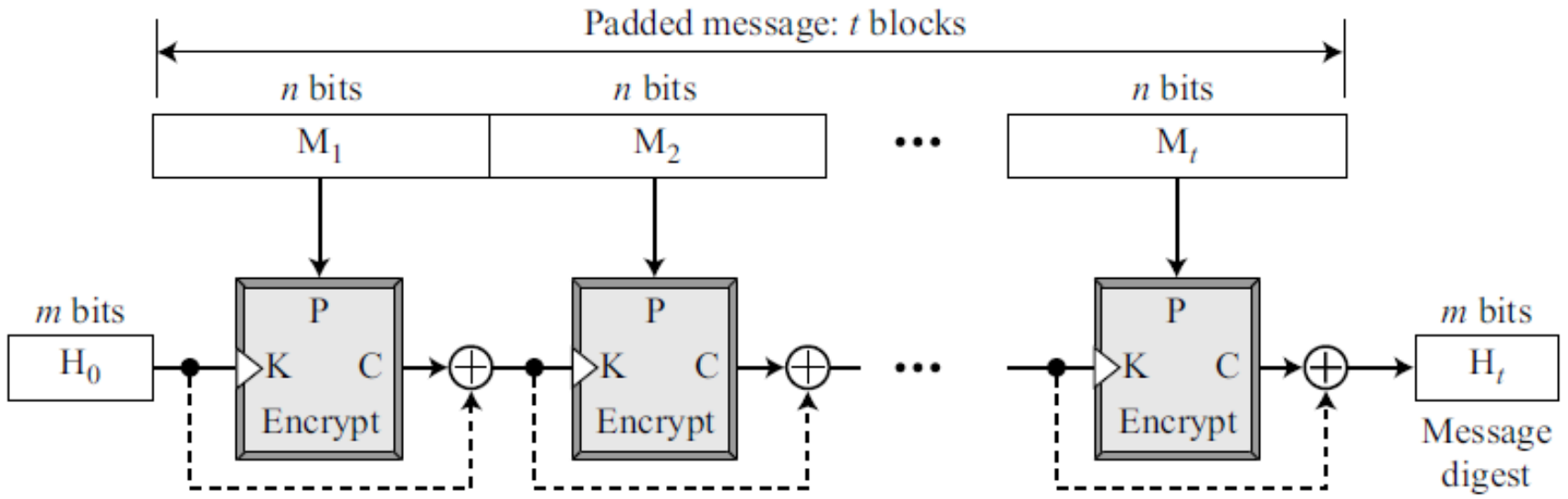
Davies-Meyer Scheme

- Same as the Rabin scheme except that it uses forward feed to protect against meet-in-the-middle attack



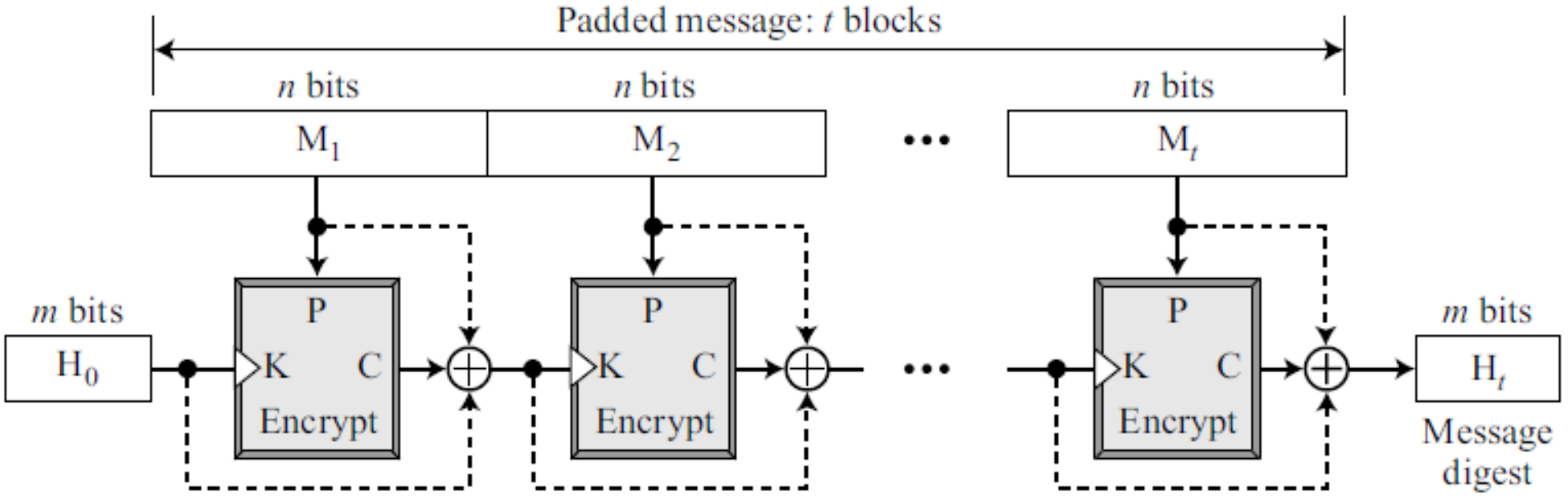
Matyas-Meyer-Oseas Scheme

- The Matyas-Meyer-Oseas scheme is a dual version of the Davies-Meyer scheme
- the previous digest is used as the key to the cryptosystem.
- The scheme can be used if the data block and the cipher key are the same size



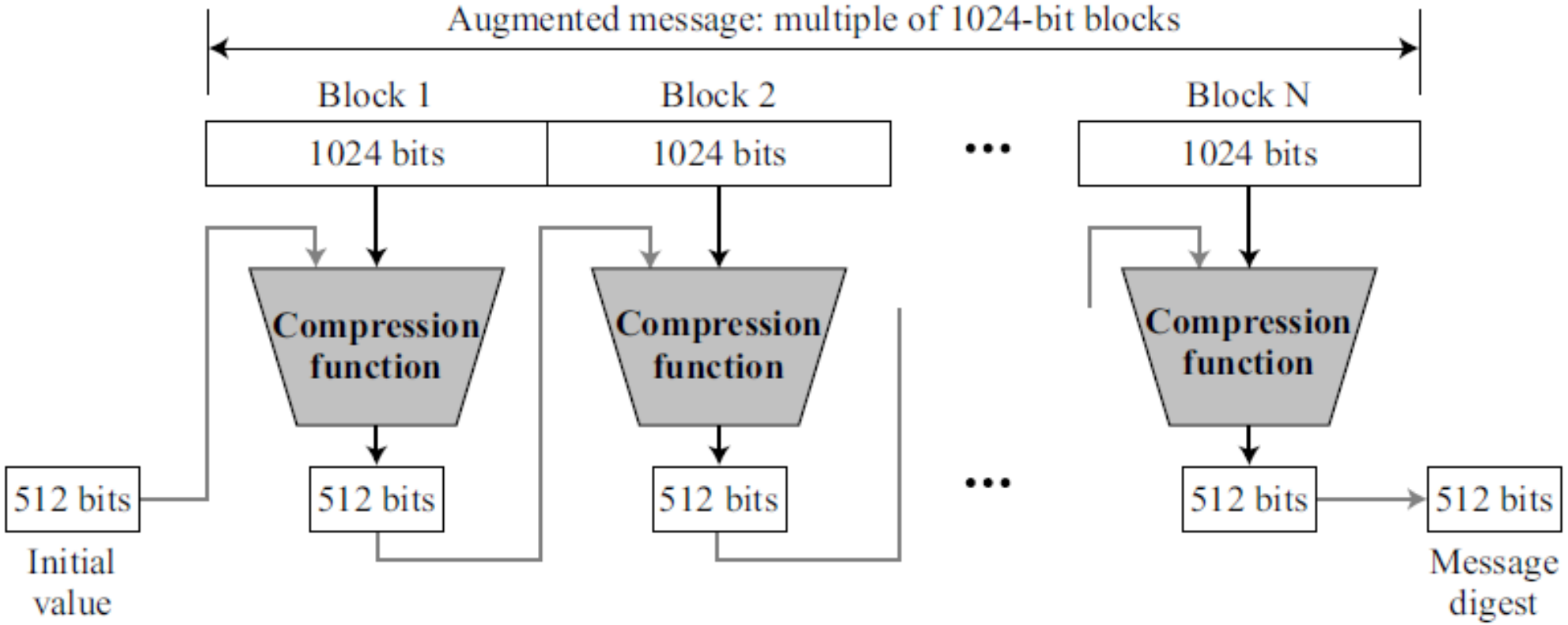
Miyaguchi-Preneel Scheme

- The Miyaguchi-Preneel scheme is an extended version of Matyas-Meyer-Oseas.
- To make the algorithm stronger against attack, the plaintext, the cipher key, and the ciphertext are all exclusive-ored together to create the new digest.
- This is the scheme used by the Whirlpool hash function



SHA-512

- 512-bit message digest based on Merkle-Damgard scheme
- SHA-512 creates a digest of 512 bits from a multiple-block message. Each block is 1024 bits in length
- The digest is initialized to a predetermined value of 512 bits



- The algorithm mixes this initial value with the first block of the message to create the first intermediate message digest of 512 bits.
- This digest is then mixed with the second block to create the second intermediate digest.
- Finally, the (N − 1)th digest is mixed with the Nth block to create the Nth digest.
- When the last block is processed, the resulting digest is the message digest for the entire message.

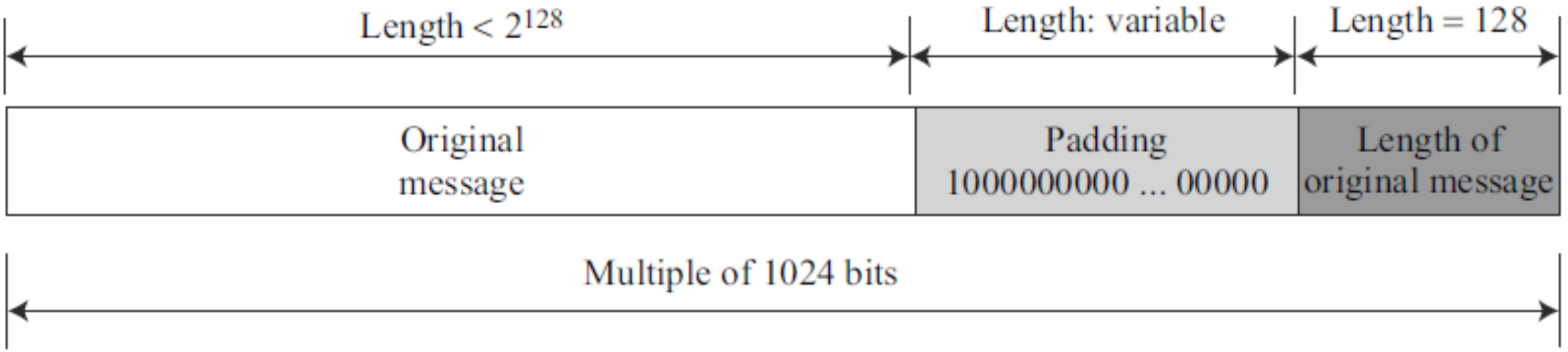
Message Preparation

- length of the original message must be less than 2^{128} bits
- if the length of a message is equal to or greater than 2^{128}, it will not be processed by SHA-512

Length Field and Padding

- SHA-512 requires the addition of a 128-bit unsigned-integer length field to the message that defines the length of the message in bits.
- This is the length of the original message before padding.
- An unsigned integer field of 128 bits can define a number between 0 and $2^{128} - 1$, which is the maximum length of the message allowed in SHA-512.
- The length field defines the length of the original message before adding the length field or the padding

Padding and length field in SHA-512



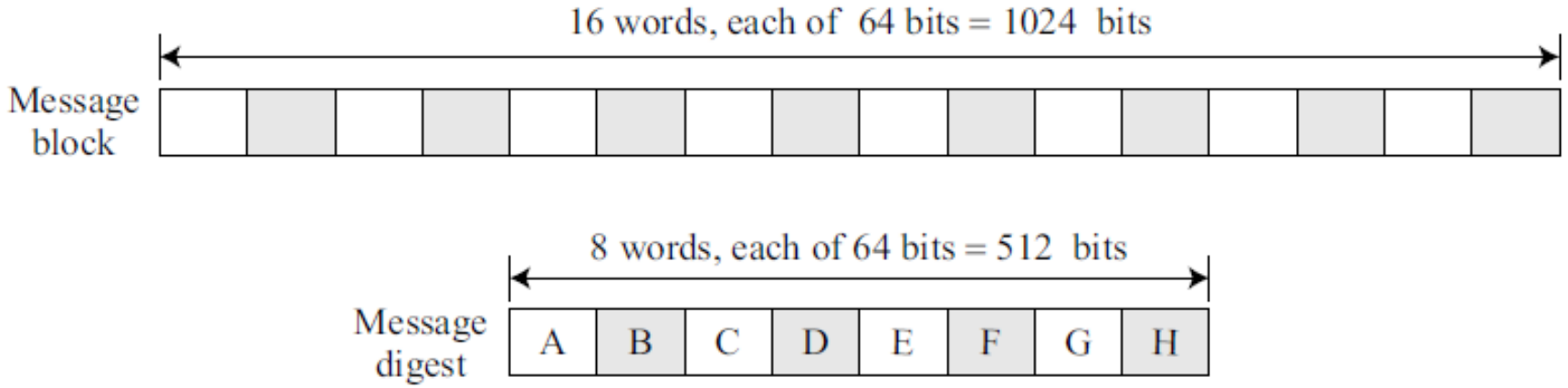
- Before the addition of the length field, the original message is padded to make the length a multiple of 1024.
- 128 bits reserved for the length field
- The length of the padding field can be calculated as follows. Let |M| be the length of the original message and |P| be the length of the padding field

$$(|M| + |P| + 128) = 0 \bmod 1024 \quad \rightarrow \quad |P| = (-|M| - 128) \bmod 1024$$

- The format of the padding is one 1 followed by the necessary number of 0s.

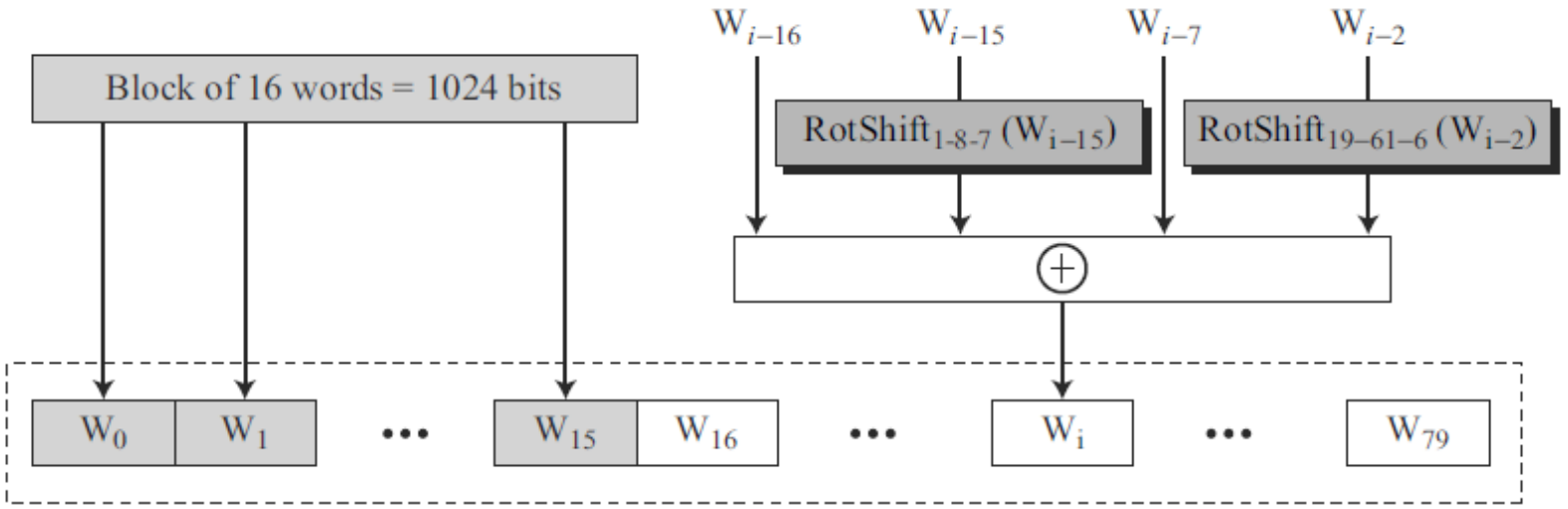
Words

- SHA-512 operates on words of 64 bits.
- after the padding and the length field are added to the message, each block of the message consists of sixteen 64-bit words
- The message digest is also made of 64-bit words, but the message digest is only eight words and the words are named A, B,C, D, E, F, G, and H



Word Expansion

- Before processing, each message block must be expanded
- A block is made of 1024 bits, or sixteen 64-bit words
- Needs 80 words in the processing phase
- So the 16-word block needs to be expanded to 80 words, from W0 to W79
- The 1024-bit block becomes the first 16 words
- Rest of the words come from already-made words according to the operation



RotShift$_{1\text{-}m\text{-}n}$ (x): $RotR_l(x) \oplus RotR_m(x) \oplus ShL_n(x)$

$RotR_i(x)$: Right-rotation of the argument x by i bits
$ShL_i(x)$: Shift-left of the argument x by i bits and padding the left by 0's.

Message Digest Initialization

- The algorithm uses eight constants A0 to H0 for message digest initialization

Buffer	*Value (in hexadecimal)*	*Buffer*	*Value (in hexadecimal)*
A_0	6A09E667F3BCC908	E_0	510E527FADE682D1
B_0	BB67AE8584CAA73B	F_0	9B05688C2B3E6C1F
C_0	3C6EF372EF94F82B	G_0	1F83D9ABFB41BD6B
D_0	A54FE53A5F1D36F1	H_0	5BE0CD19137E2179

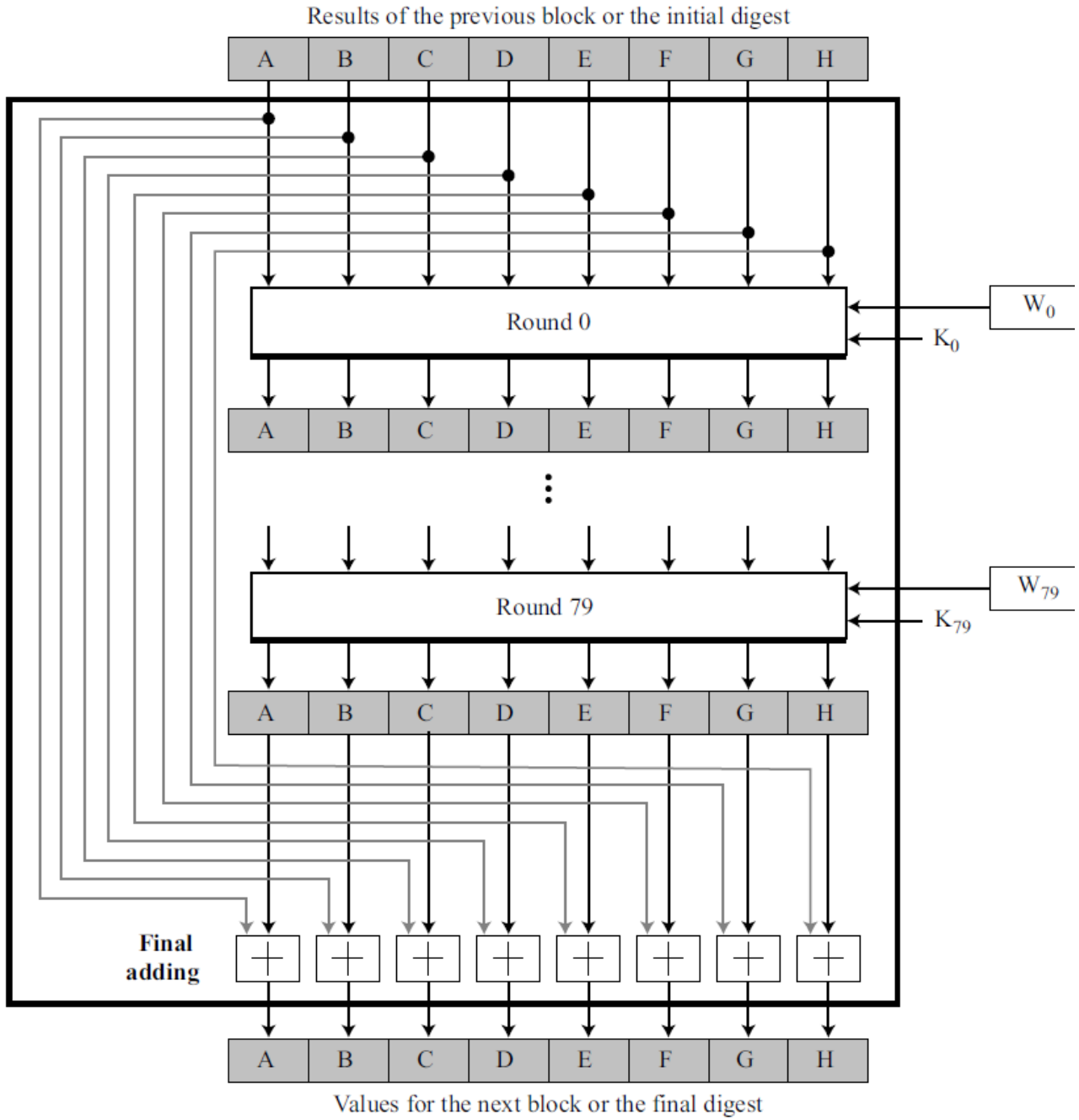
- The values are calculated from the first eight prime numbers (2, 3, 5, 7, 11, 13, 17, and 19)
- Each value is the fraction part of the square root of the corresponding prime number after converting to binary and keeping only the first 64 bits
- eighth prime is 19, with the square root $(19)^{1/2} = 4.35889894354$, converting the number to binary with only 64 bits in the fraction part

$$(100.0101\ 1011\ 1110\ \ldots\ 1001)_2 \quad \rightarrow \quad (4.5BE0CD19137E2179)_{16}$$

- SHA-512 keeps the fraction part, (5BE0CD19137E2179)16, as an unsigned integer.

Compression Function

- SHA-512 creates a 512-bit (eight 64-bit words) message digest from a multiple-block message where each block is 1024 bits
- The processing of each block of data in SHA-512 involves 80 rounds.
- In each round, the contents of eight previous buffers, one word from the expanded block (Wi), and one 64-bit constant (Ki) are mixed together and then operated on to create a new set of eight buffers
- At the beginning of processing, the values of the eight buffers are saved into eight temporary variables.
- At the end of the processing (after step 79), these values are added to the values created from step 79. this last operation is called the final adding

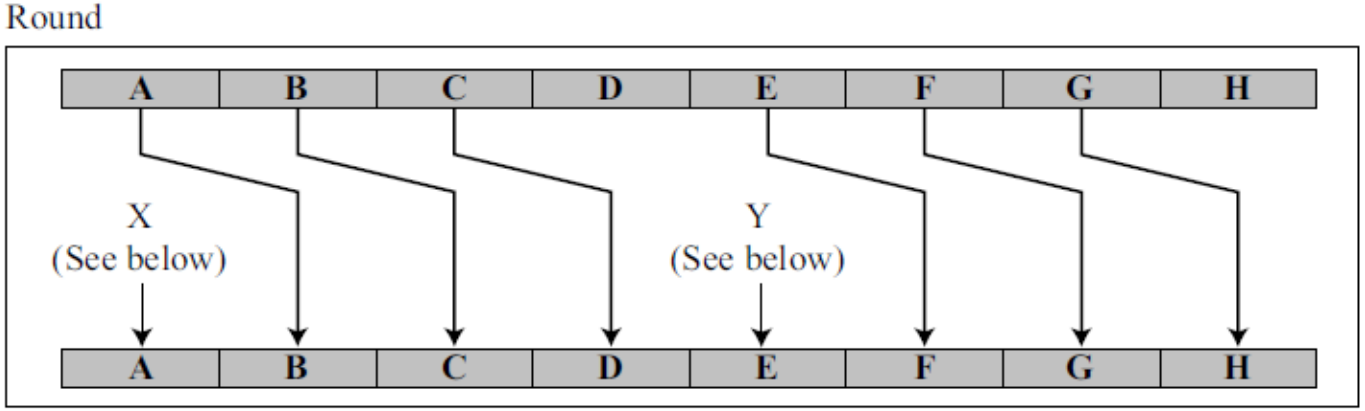

Results of the previous block or the initial digest
A
B
C
D
E
F
G
H
Round 0
W_0
K_0
Round 79
W_{79}
K_{79}
Final adding
Values for the next block or the final digest

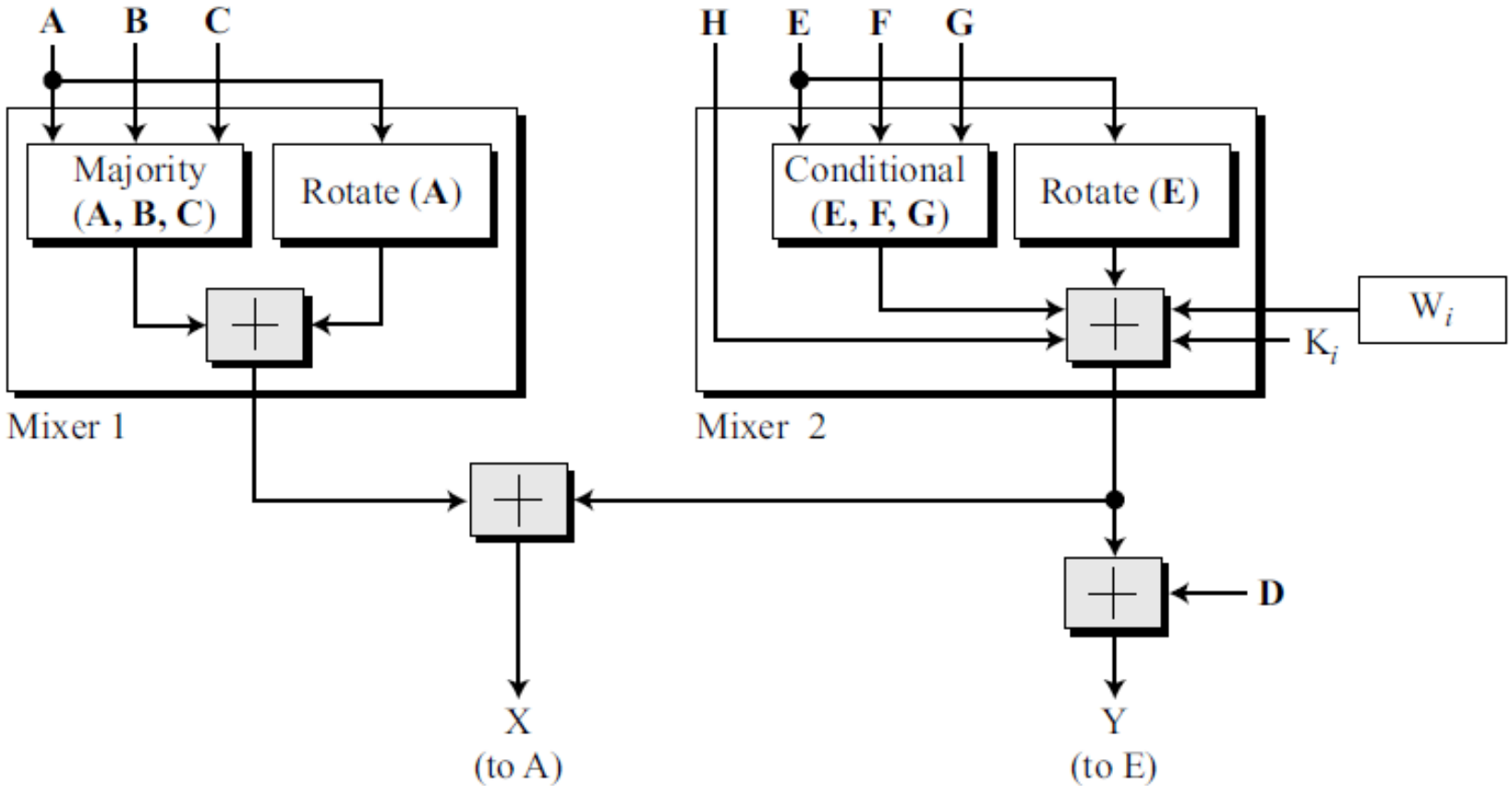
Structure of Each Round

- eight new values for the 64-bit buffers are created from the values of the buffers in the previous round
- six buffers are the exact copies of one of the buffers in the previous round as shown below

$$A \rightarrow B \quad B \rightarrow C \quad C \rightarrow D \quad E \rightarrow F \quad F \rightarrow G \quad G \rightarrow H$$

- Two of the new buffers, A and E, receive their inputs from some complex functions that involve some of the previous buffers, the corresponding word for this round (Wi), and the corresponding constant for this round (Ki).





Majority (x, y, z)

$(x \text{ AND } y) \oplus (y \text{ AND } z) \oplus (z \text{ AND } x)$

Conditional (x, y, z)

$(x \text{ AND } y) \oplus (\text{NOT } x \text{ AND } z)$

Rotate (x)

$RotR_{28}(x) \oplus RotR_{34}(x) \oplus RotR_{39}(x)$

$\boxplus$ addition modulo 2^{64}

$RotR_i(x)$: Right-rotation of the argument x by i bits

There are two mixers, three functions, and several operators. Each mixer combines two functions. The description of the functions and operators follows:

1. The Majority function, as we call it, is a bitwise function. It takes three corresponding bits in three buffers (A, B, and C) and calculates

(Aj AND Bj) $\oplus$ (Bj AND Cj) $\oplus$ (Cj AND Aj)

The resulting bit is the majority of three bits. If two or three bits are 1's, the resulting bit is 1; otherwise it is 0.

2. The Conditional function is also a bitwise function. It takes three corresponding bits in three buffers (E, F, and G) and calculates

(Ej AND Fj) $\oplus$ (NOT Ej AND Gj)

The resulting bit is the logic "If Ej then Fj; else Gj".

3. The Rotate function, right-rotates the three instances of the same buffer (A or E) and applies the exclusive-or operation on the results.

Rotate (A): RotR28(A) $\oplus$ RotR34(A) $\oplus$ RotR29(A)

Rotate (E): RotR28(E) $\oplus$ RotR34(E) $\oplus$ RotR29(E)

4. The right-rotation function, RotRi(x), right-rotates its argument i bits; it is actually a circular shiftright operation.

5. The addition operator used in the process is addition modulo 2^{64}. This means that the result of adding two or more buffers is always a 64-bit word.

6. There are 80 constants, K0 to K79, each of 64 bits as shown in Table in hexadecimal format (four in a row). Similar to the initial values for the eight digest buffers, these values are calculated from the first 80 prime numbers (2, 3,…, 409).

428A2F98D728AE22	7137449123EF65CD	B5C0FBCFEC4D3B2F	E9B5DBA58189DBBC
3956C25BF348B538	59F111F1B605D019	923F82A4AF194F9B	AB1C5ED5DA6D8118
D807AA98A3030242	12835B0145706FBE	243185BE4EE4B28C	550C7DC3D5FFB4E2
72BE5D74F27B896F	80DEB1FE3B1696B1	9BDC06A725C71235	C19BF174CF692694
E49B69C19EF14AD2	EFBE4786384F25E3	0FC19DC68B8CD5B5	240CA1CC77AC9C65
2DE92C6F592B0275	4A7484AA6EA6E483	5CB0A9DCBD41FBD4	76F988DA831153B5
983E5152EE66DFAB	A831C66D2DB43210	B00327C898FB213F	BF597FC7BEEF0EE4
C6E00BF33DA88FC2	D5A79147930AA725	06CA6351E003826F	142929670A0E6E70
27B70A8546D22FFC	2E1B21385C26C926	4D2C6DFC5AC42AED	53380D139D95B3DF
650A73548BAF63DE	766A0ABB3C77B2A8	81C2C92E47EDAEE6	92722C851482353B
A2BFE8A14CF10364	A81A664BBC423001	C24B8B70D0F89791	C76C51A30654BE30
D192E819D6EF5218	D69906245565A910	F40E35855771202A	106AA07032BBD1B8
19A4C116B8D2D0C8	1E376C085141AB53	2748774CDF8EEB99	34B0BCB5E19B48A8
391C0CB3C5C95A63	4ED8AA4AE3418ACB	5B9CCA4F7763E373	682E6FF3D6B2B8A3
748F82EE5DEFB2FC	78A5636F43172F60	84C87814A1F0AB72	8CC702081A6439EC
90BEFFFA23631E28	A4506CEBDE82BDE9	BEF9A3F7B2C67915	C67178F2E372532B
CA273ECEEA26619C	D186B8C721C0C207	EADA7DD6CDE0EB1E	F57D4F7FEE6ED178
06F067AA72176FBA	0A637DC5A2C898A6	113F9804BEF90DAE	1B710B35131C471B
28DB77F523047D84	32CAAB7B40C72493	3C9EBE0A15C9BEBC	431D67C49C100D4C
4CC5D4BECB3E42B6	4597F299CFC657E2	5FCB6FAB3AD6FAEC	6C44198C4A475817

- Each value is the fraction part of the cubic root of the corresponding prime number after converting it to binary and keeping only the first 64 bits
- 80th prime is 409, with the cubic root (409)1/3 = 7.42291412044. Converting this number to binary with only 64 bits in the fraction part, we get (111.0110 1100 0100 0100 . . . 0111)2 → (7.6C44198C4A475817)16
- SHA-512 keeps the fraction part, (6C44198C4A475817)16, as an unsigned integer.

What is the number of padding bits if the length of the original message is 2590 bits?

Solution

the number of padding bits is calculated as follows:

$$|P| = (-2590 - 128) \bmod 1024 = -2718 \bmod 1024 = 354$$

The padding consists of one 1 followed by 353 0's.

Do we need padding if the length of the original message is already a multiple of 1024 bits?

Yes, because we need to add the length field. So padding is needed to make the new block a multiple of 1024 bits

Example

State the value of the padding field in SHA-512 if the length of the message is

a. 1919 bits

b. 1920 bits

c. 1921 bits

State the value of the length field in SHA-512 if the length of the message is

a. 1919 bits

b. 1920 bits

c. 1921 bits

Athe Majority function on buffers A, B, and C. If the leftmost hexadecimal digits of these buffers are 0x7, 0xA, and 0xE, respectively, what is the leftmost digit of the result?

Solution

The digits in binary are 0111, 1010, and 1110.

a. The first bits are 0, 1, and 1. The majority is 1.

using the definition of the Majority function:

$$(0 \text{ AND } 1) \oplus (1 \text{ AND } 1) \oplus (1 \text{ AND } 0) = 0 \oplus 1 \oplus 0 = 1$$

b. The second bits are 1, 0, and 1. The majority is 1.

c. The third bits are 1, 1, and 1. The majority is 1.

d. The fourth bits are 1, 0, and 0. The majority is 0.

The result is 1110, or 0xE in hexadecimal.

Example

Apply the Conditional function on E, F, and G buffers. If the leftmost hexadecimal digits of these buffers are 0x9, 0xA, and 0xF respectively, what is the leftmost digit of the result?

The digits in binary are 1001, 1010, and 1111.

a. The first bits are 1, 1, and 1. Since E1 = 1, the result is F1, which is 1. We can also use the

$$(1 \text{ AND } 1) \oplus (\text{NOT } 1 \text{ AND } 1) = 1 \oplus 0 = 1$$

definition of the Condition function to prove the result:

b. The second bits are 0, 0, and 1. Since E2 is 0, the result is G2, which is 1.

c. The third bits are 0, 1, and 1. Since E3 is 0, the result is G3, which is 1.

d. The fourth bits are 1, 0, and 1. Since E4 is 1, the result is F4, which is 0.

The result is 1110, or 0xE in hexadecimal.

Example

Text – abc

01100001 01100010 01100011

message is padded to a length congruent to 896 modulo 1024

the padding consists of , consisting of a “1” bit followed by 871 “0” bits.

Then a 128-bit length value is appended to the message, which contains the length of the original message (before the padding)

The original length is 24 bits, or a hexadecimal value of 18.

```
61626380000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000018
```

This block is assigned to the words W0 – W15 of the message schedule

$$W_0 = 6162638000000000$$
$$W_1 = 0000000000000000$$
$$W_2 = 0000000000000000$$
$$W_3 = 0000000000000000$$
$$W_4 = 0000000000000000$$
$$W_5 = 0000000000000000$$
$$W_6 = 0000000000000000$$
$$W_7 = 0000000000000000$$
$$W_8 = 0000000000000000$$
$$W_9 = 0000000000000000$$
$$W_{10} = 0000000000000000$$
$$W_{11} = 0000000000000000$$
$$W_{12} = 0000000000000000$$
$$W_{13} = 0000000000000000$$
$$W_{14} = 0000000000000000$$
$$W_{15} = 0000000000000018$$

a	6a09e667f3bcc908
b	bb67ae8584caa73b
c	3c6ef372fe94f82b
d	a54ff53a5f1d36f1
e	510e527fade682d1
f	9b05688c2b3e6c1f
g	1f83d9abfb41bd6b
h	5be0cd19137e2179

f6afceb8bcfcddf5	1320f8c9fb872cc0
6a09e667f3bcc908	f6afceb8bcfcddf5
bb67ae8584caa73b	6a09e667f3bcc908
3c6ef372fe94f82b	bb67ae8584caa73b
58cb02347ab51f91	c3d4ebfd48650ffa
510e527fade682d1	58cb02347ab51f91
9b05688c2b3e6c1f	510e527fade682d1
1f83d9abfb41bd6b	9b05688c2b3e6c1f

73a54f399fa4b1b2 10d9c4c4295599f6 d67806db8b148677 654ef9abec389ca9 d08446aa79693ed7 9bb4d39778c07f9e 25c96a7768fb2aa3 ceb9fc3691ce8326

The hash value is then calculated as

$$H_{1,0} = \texttt{6a09e667f3bcc908} + \texttt{73a54f399fa4b1b2} = \texttt{ddaf35a193617aba}$$

$$H_{1,1} = \texttt{bb67ae8584caa73b} + \texttt{10d9c4c4295599f6} = \texttt{cc417349ae204131}$$

$$H_{1,2} = \texttt{3c6ef372fe94f82b} + \texttt{d67806db8b148677} = \texttt{12e6fa4e89a97ea2}$$

$$H_{1,3} = \texttt{a54ff53a5f1d36f1} + \texttt{654ef9abec389ca9} = \texttt{0a9eeee64b55d39a}$$

$$H_{1,4} = \texttt{510e527fade682d1} + \texttt{d08446aa79693ed7} = \texttt{2192992a274fc1a8}$$

$$H_{1,5} = \texttt{9b05688c2b3e6c1f} + \texttt{9bb4d39778c07f9e} = \texttt{36ba3c23a3feebbd}$$

$$H_{1,6} = \texttt{1f83d9abfb41bd6b} + \texttt{25c96a7768fb2aa3} = \texttt{454d4423643ce80e}$$

$$H_{1,7} = \texttt{5be0cd19137e2179} + \texttt{ceb9fc3691ce8326} = \texttt{2a9ac94fa54ca49f}$$

The resulting 512-bit message digest is

```
ddaf35a193617aba cc417349ae204131 12e6fa4e89a97ea2 0a9eeee64b55d39a
2192992a274fc1a8 36ba3c23a3feebbd 454d4423643ce80e 2a9ac94fa54ca49f
```

Suppose now that we change the input message by one bit, from “abc” to “cbc”.

Then, the 1024-bit message block is

6362638000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000018

And the resulting 512-bit message digest is

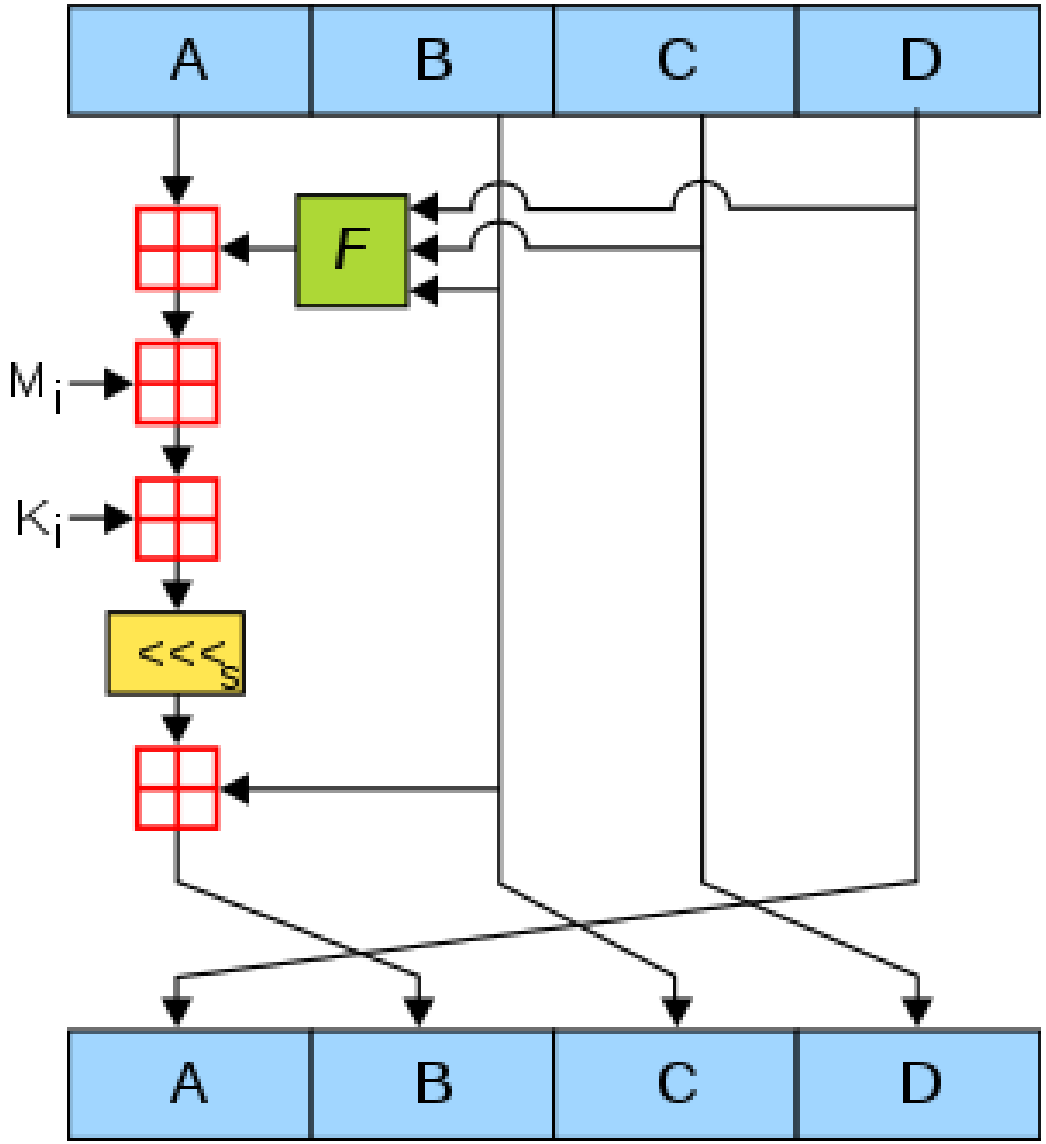
531668966ee79b70 0b8e593261101354 4273f7ef7b31f279 2a7ef68d53f93264
319c165ad96d9187 55e6a204c2607e27 6e05cdf993a64c85 ef9e1e125c0f925f

The number of bit positions that differ between the two hash values is 253, almost exactly half the bit positions, indicating that SHA-512 has a good avalanche effect.

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 (sixteen 32-bit words);
- the message is padded so that its length is divisible by 512.
- The padding works as follows: first, a single bit, 1, is appended to the end of the message. This is followed by as many zeros as are required to bring the length of the message up to 64 bits fewer than a multiple of 512. The remaining bits are filled up with 64 bits representing the length of the original message, modulo 2^{64}.

- 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.
- The main algorithm then uses each 512-bit message block in turn to modify the state.
- The processing of a message block consists of four similar stages, termed *rounds*; each round is composed of 16 similar operations based on a non-linear function *F*, modular addition, and left rotation.
- Figure illustrates one operation within a round. There are four possible functions; a different one is used in each round:


A
B
C
D
F
M_i
K_i
<<<s
A
B
C
D

s[0..15] := { 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22 }

s[16..31] := { 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20 }

s[32..47] := { 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23 }

s[48..63] := { 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21 }

K[0.. 3] := { 0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee }
K[4.. 7] := { 0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501 }
K[8..11] := { 0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be }
K[12..15] := { 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821 }
K[16..19] := { 0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa }
K[20..23] := { 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8 }
K[24..27] := { 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed }
K[28..31] := { 0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a }
K[32..35] := { 0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c }
K[36..39] := { 0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70 }
K[40..43] := { 0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05 }
K[44..47] := { 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665 }
K[48..51] := { 0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039 }
K[52..55] := { 0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1 }
K[56..59] := { 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1 }
K[60..63] := { 0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391 }

A = 0x67452301

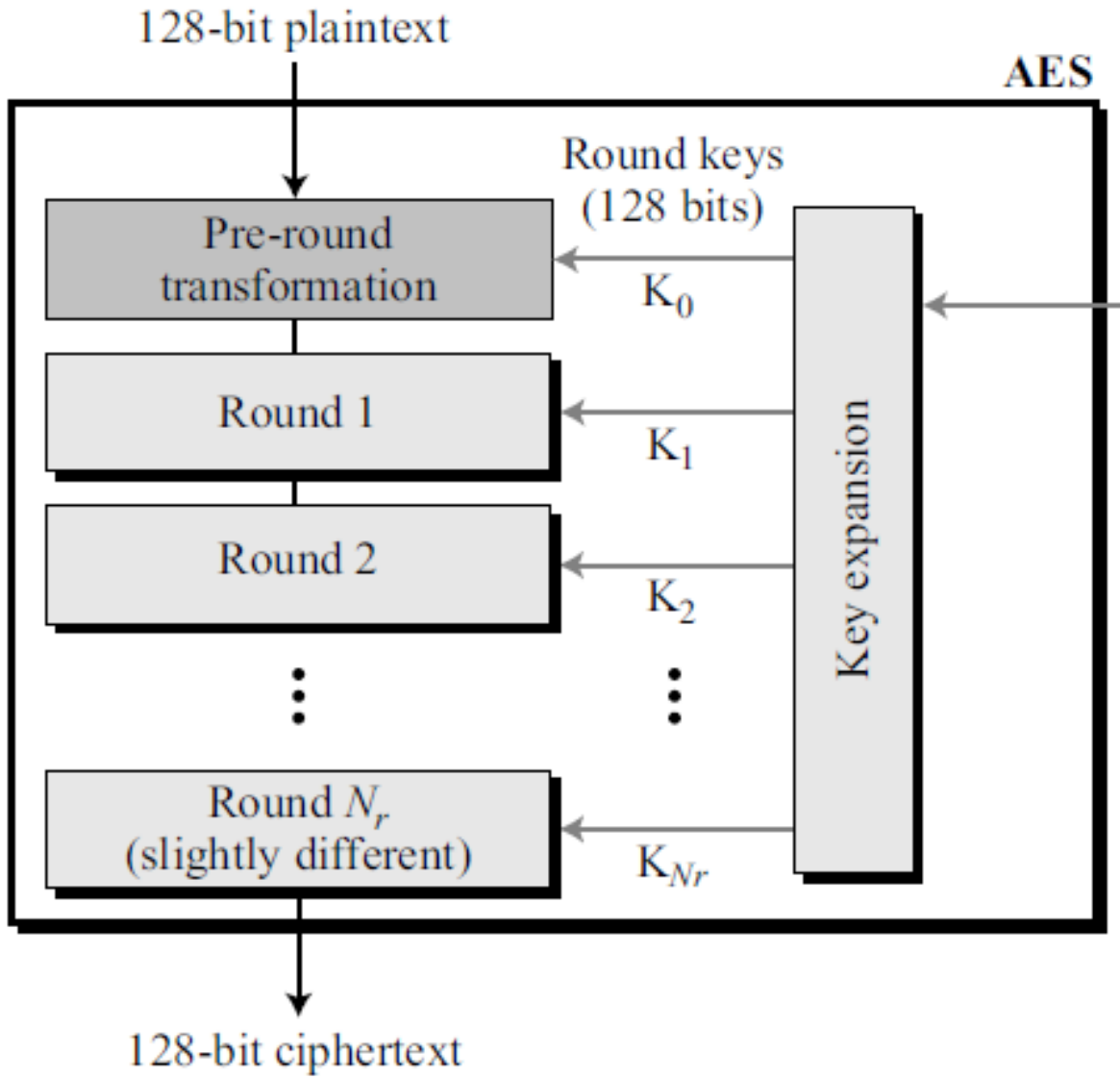
B= 0xefcdab89

C= 0x98badcfe

D= 0x10325476

$$
\begin{aligned}
F(B, C, D) &= (B \wedge C) \vee (\neg B \wedge D) \\
G(B, C, D) &= (B \wedge D) \vee (C \wedge \neg D) \\
H(B, C, D) &= B \oplus C \oplus D \\
I(B, C, D) &= C \oplus (B \vee \neg D)
\end{aligned}
$$

Overview of AES-General design of AES encryption cipher



Cipher key
(128, 192, or 256 bits)

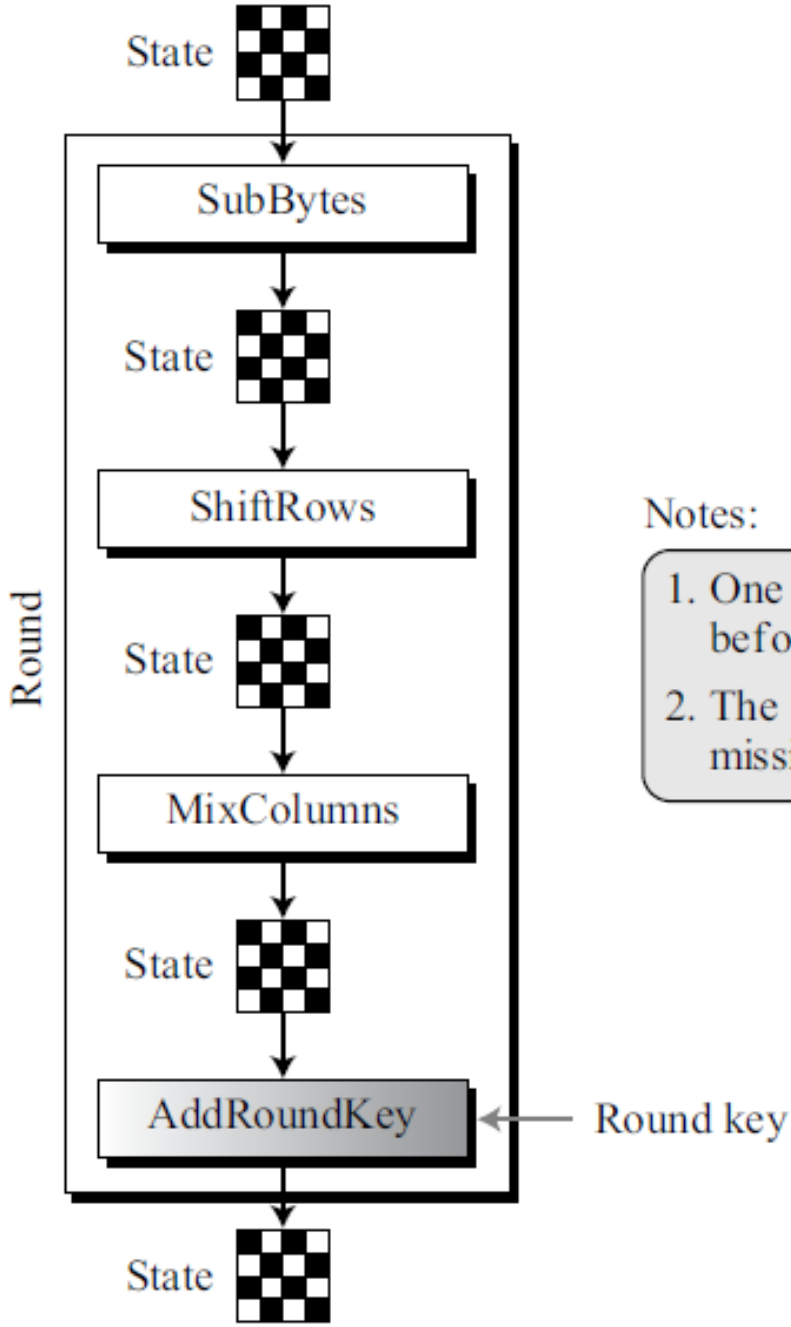
Nr	Key size
10	128
12	192
14	256

Relationship between number of rounds and cipher key size

Text	A	E	S	U	S	E	S	A	M	A	T	R	I	X	Z	Z
Hexadecimal	00	04	12	14	12	04	12	00	0C	00	13	11	08	23	19	19

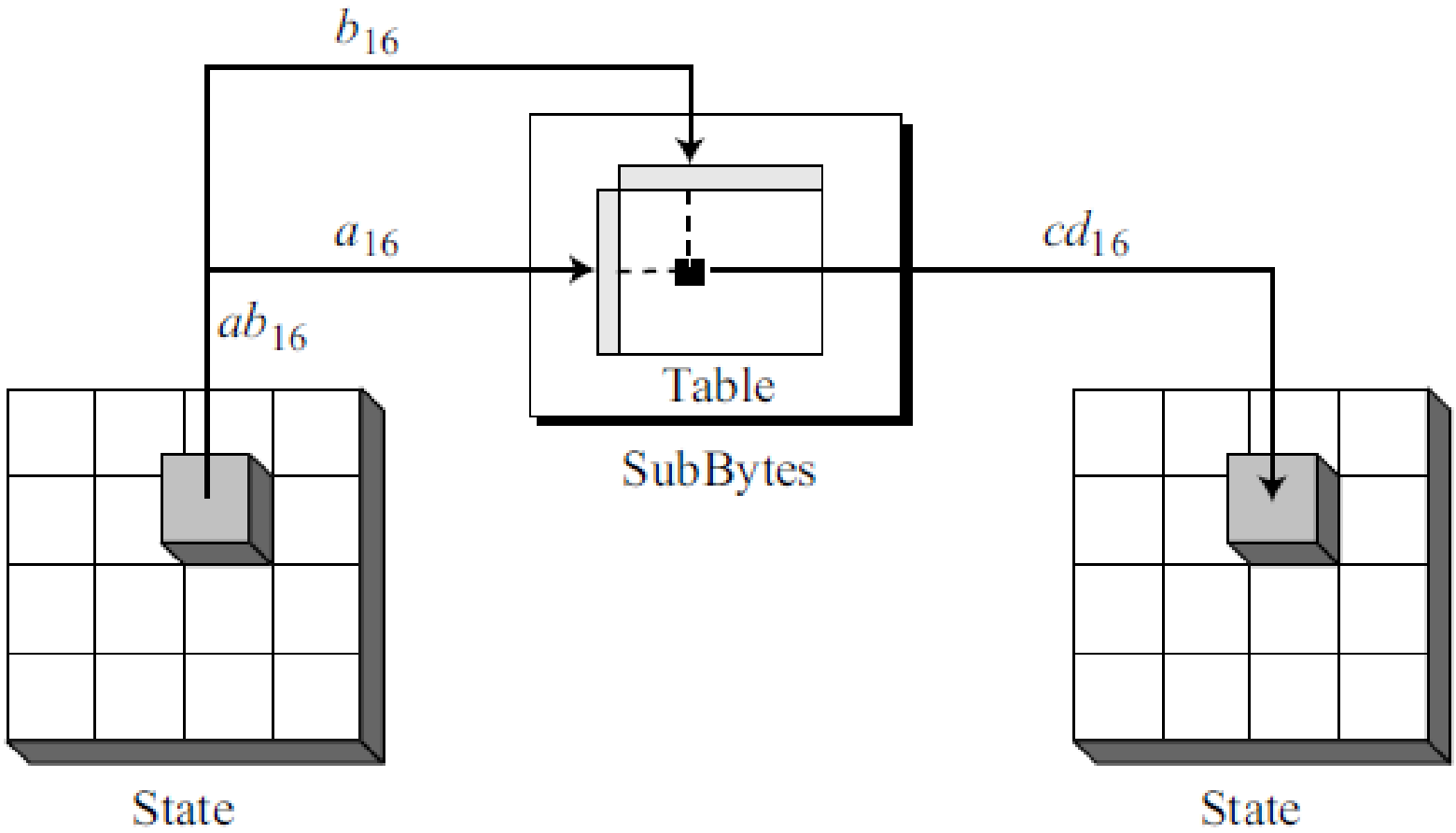
$$\begin{bmatrix} 00 & 12 & 0C & 08 \\ 04 & 04 & 00 & 23 \\ 12 & 12 & 13 & 19 \\ 14 & 00 & 11 & 19 \end{bmatrix}$$

State



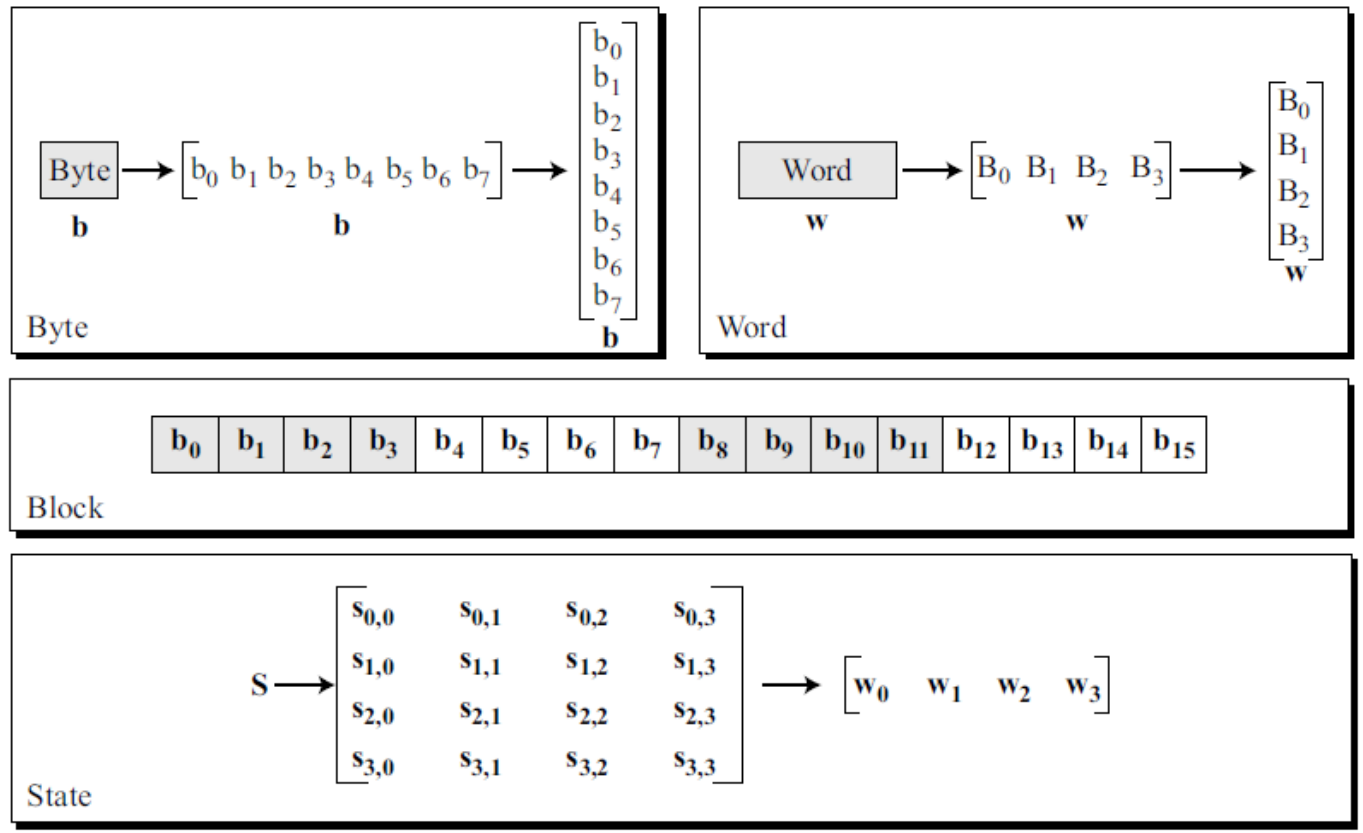
Notes:

1. One AddRoundKey is applied before the first round.
2. The third transformation is missing in the last round.

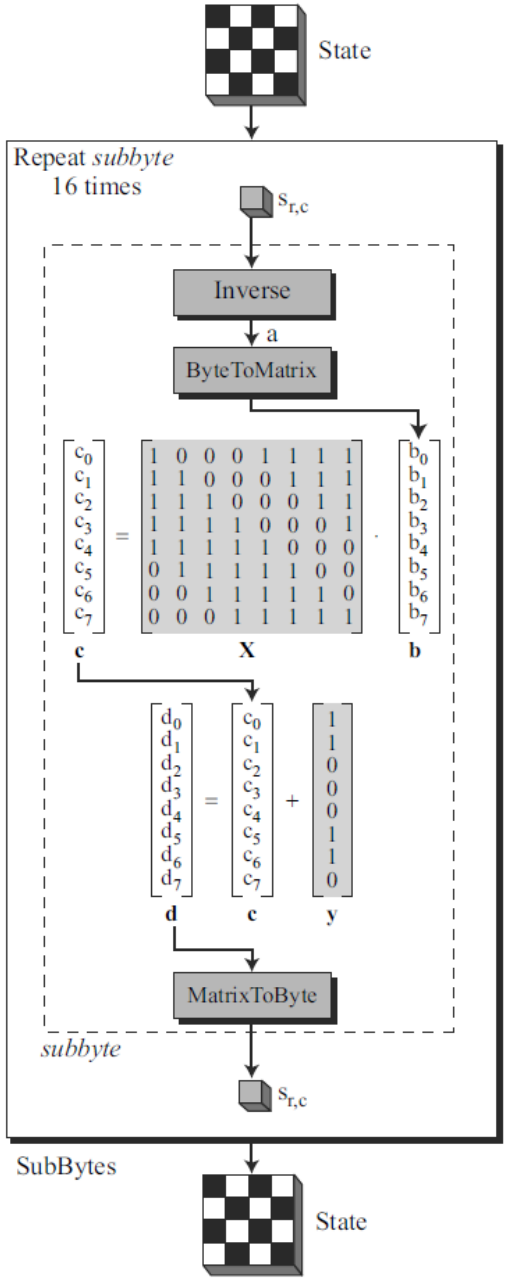

b_{16}
a_{16}
cd_{16}
ab_{16}
Table
SubBytes
State
State

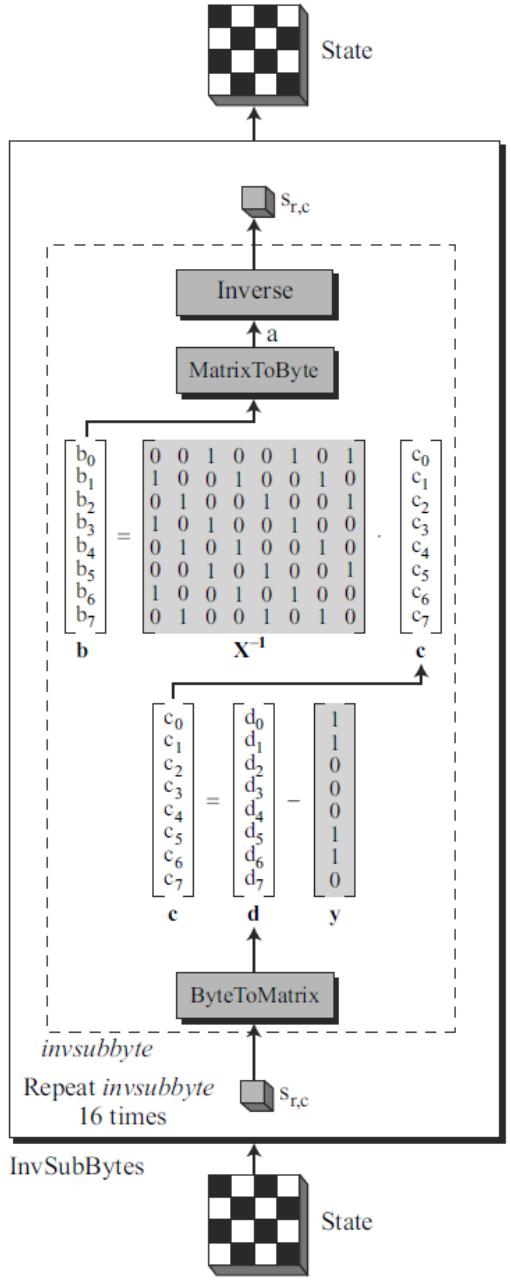
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*A*	*B*	*C*	*D*	*E*	*F*
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*A*	*B*	*C*	*D*	*E*	*F*
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	CB	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

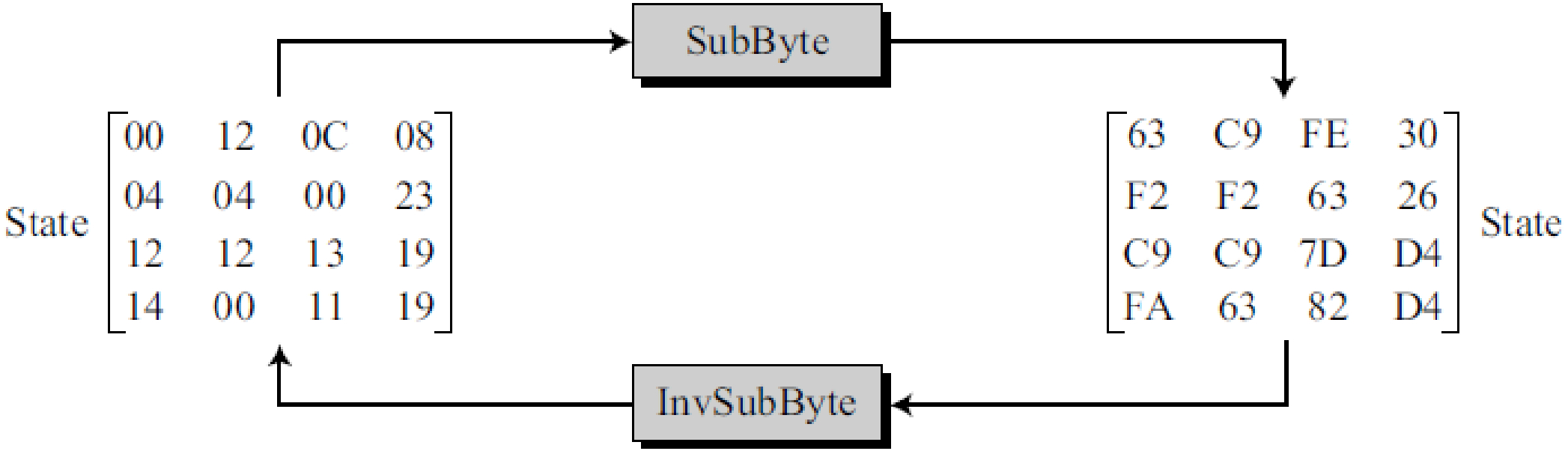

Byte
b
b0 b1 b2 b3 b4 b5 b6 b7
b
b0
b1
b2
b3
b4
b5
b6
b7
b
Byte
Word
w
B0 B1 B2 B3
w
B0
B1
B2
B3
w
Word
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15
Block
S
s0,0 s0,1 s0,2 s0,3
s1,0 s1,1 s1,2 s1,3
s2,0 s2,1 s2,2 s2,3
s3,0 s3,1 s3,2 s3,3
w0 w1 w2 w3
State

- Transformation is defined algebraically using the GF(2^8) field with the irreducible polynomials (x8 + x4 + x3+ x + 1)

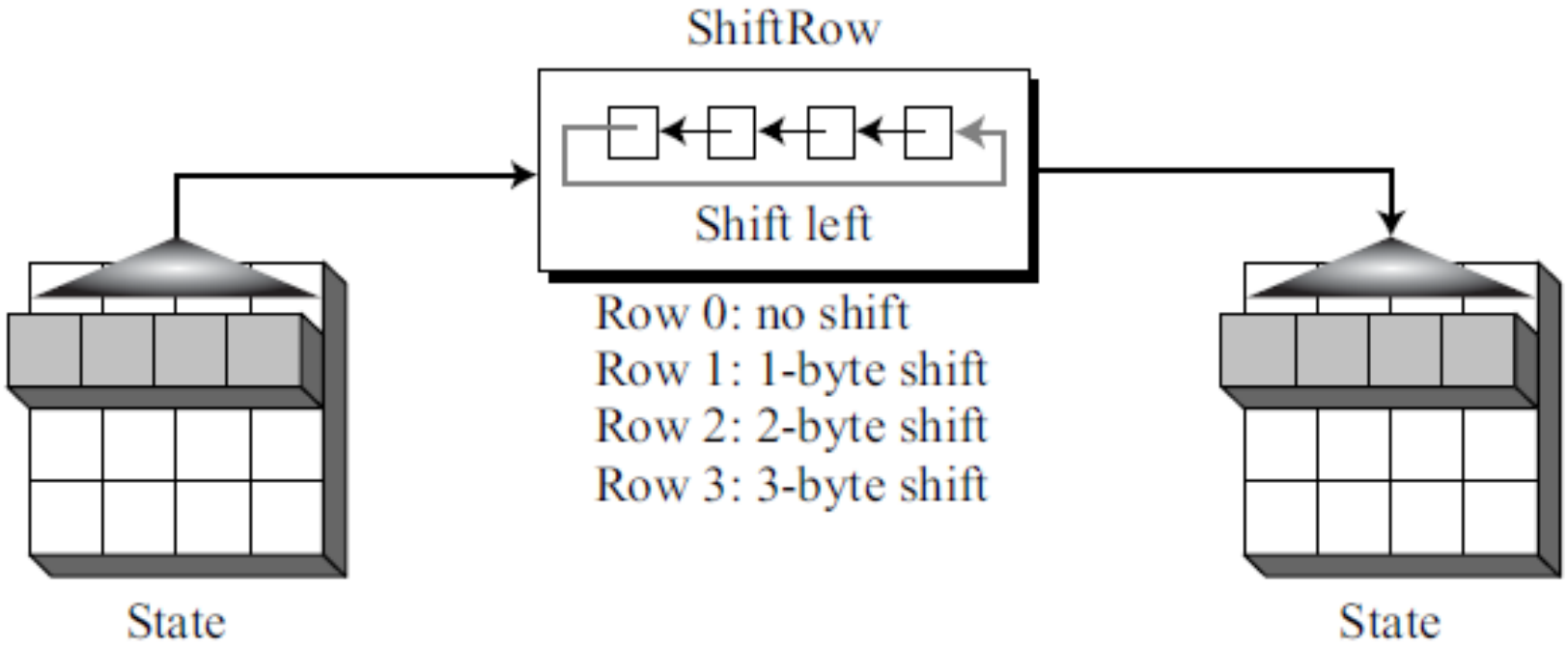




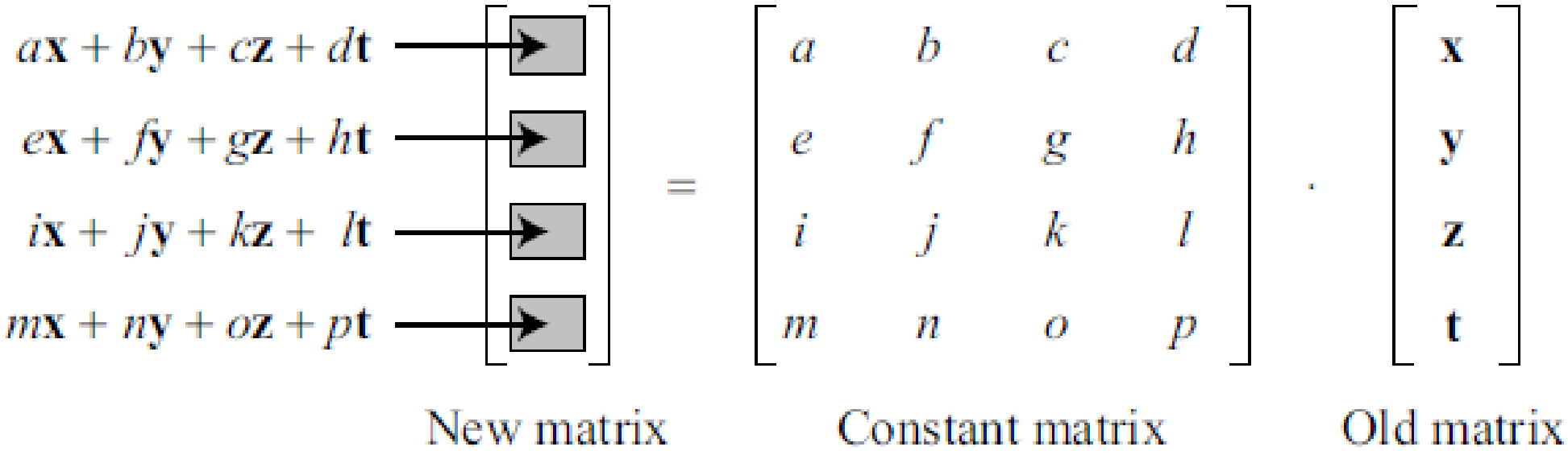
SubBytes transformations



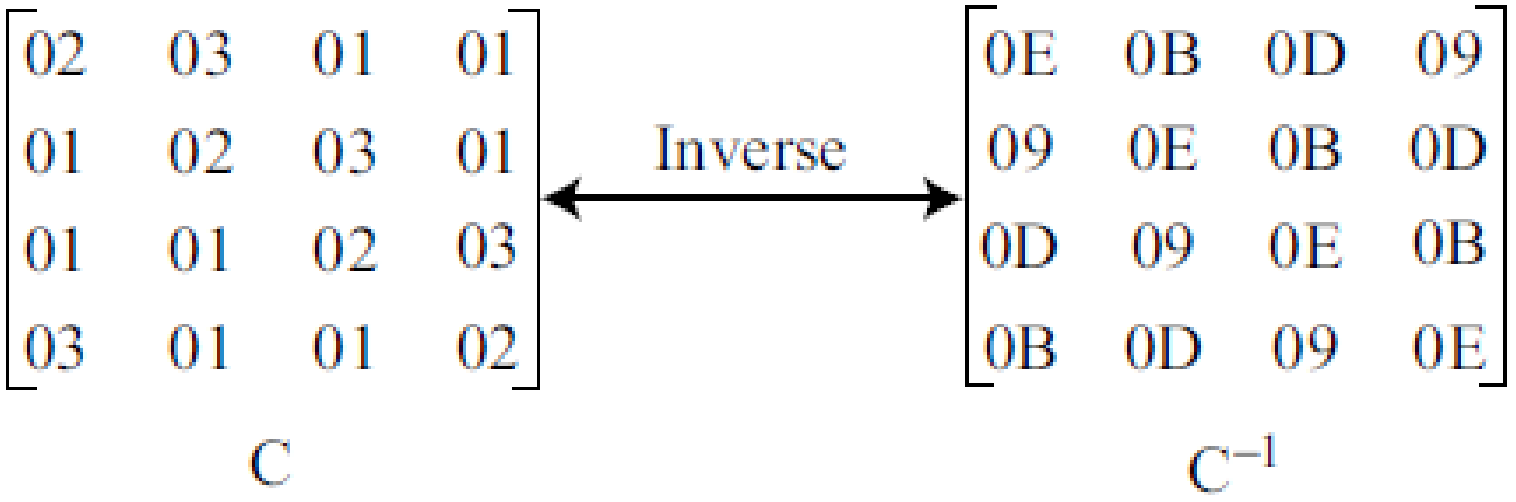
Shift Rows transformation



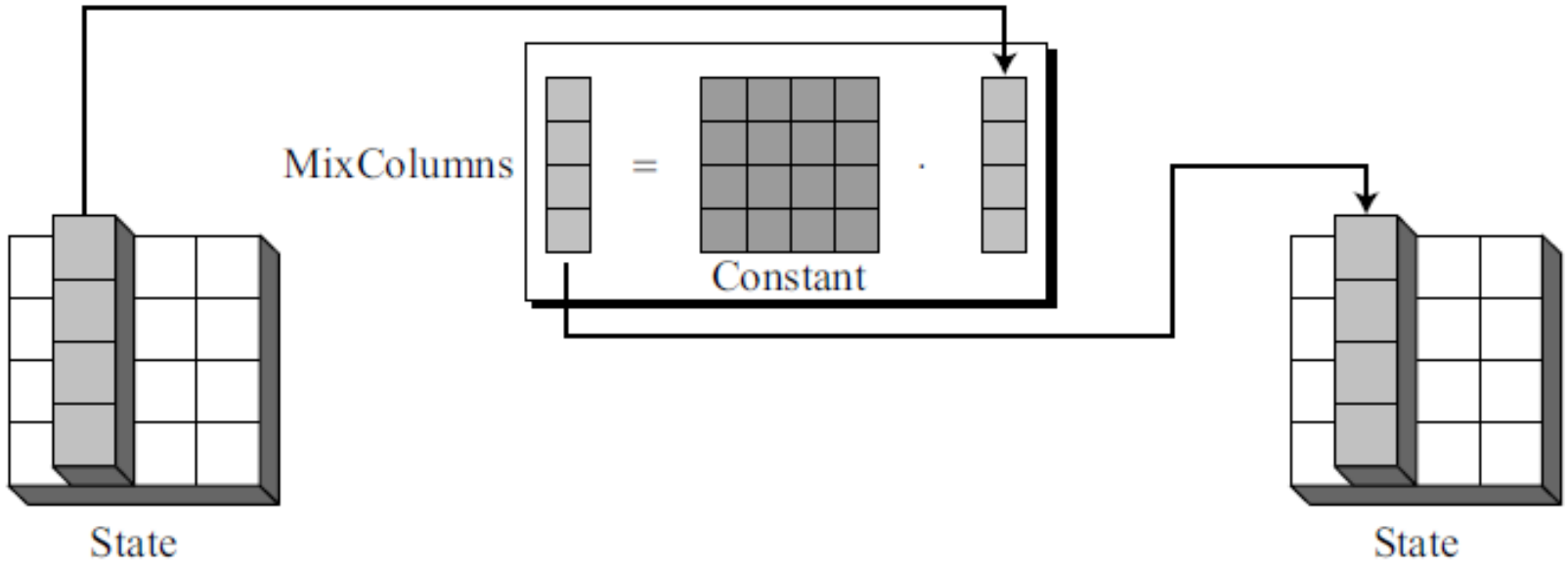
Mixing bytes using matrix multiplication



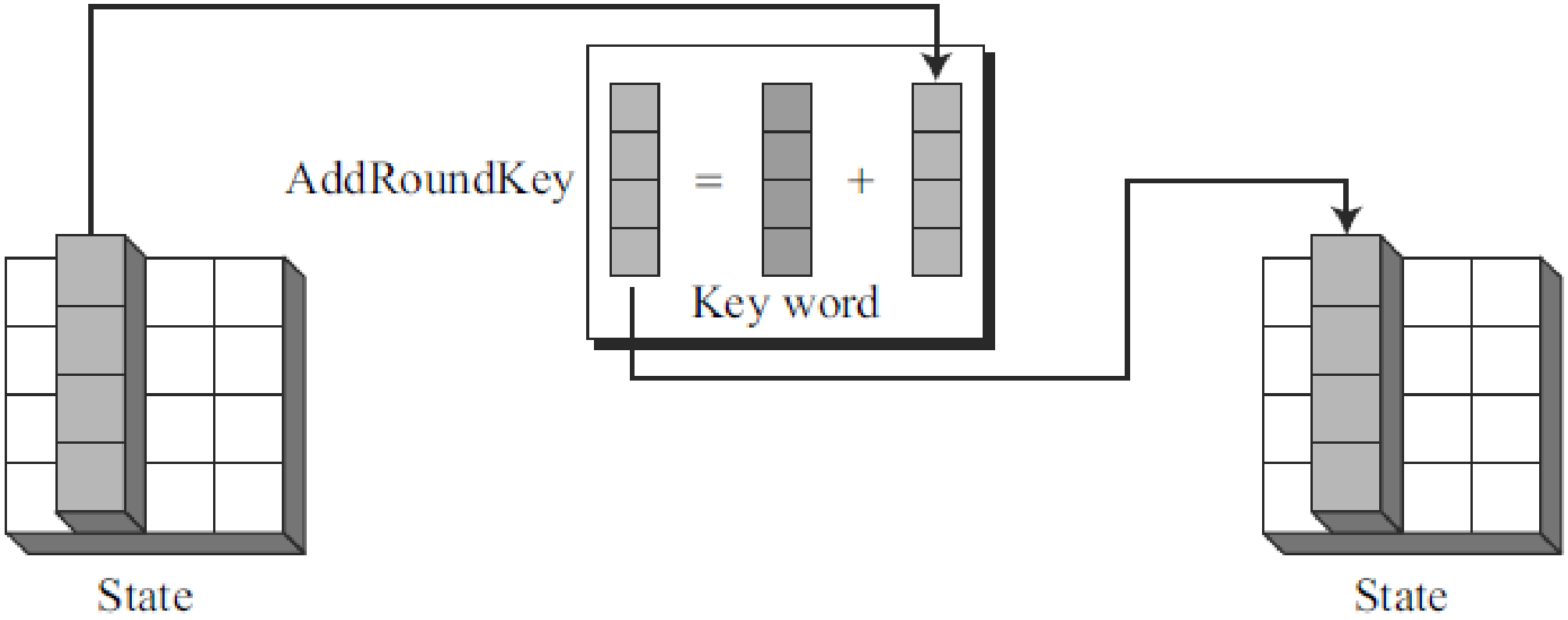
Constant matrices used by MixColumns and InvMixColumns



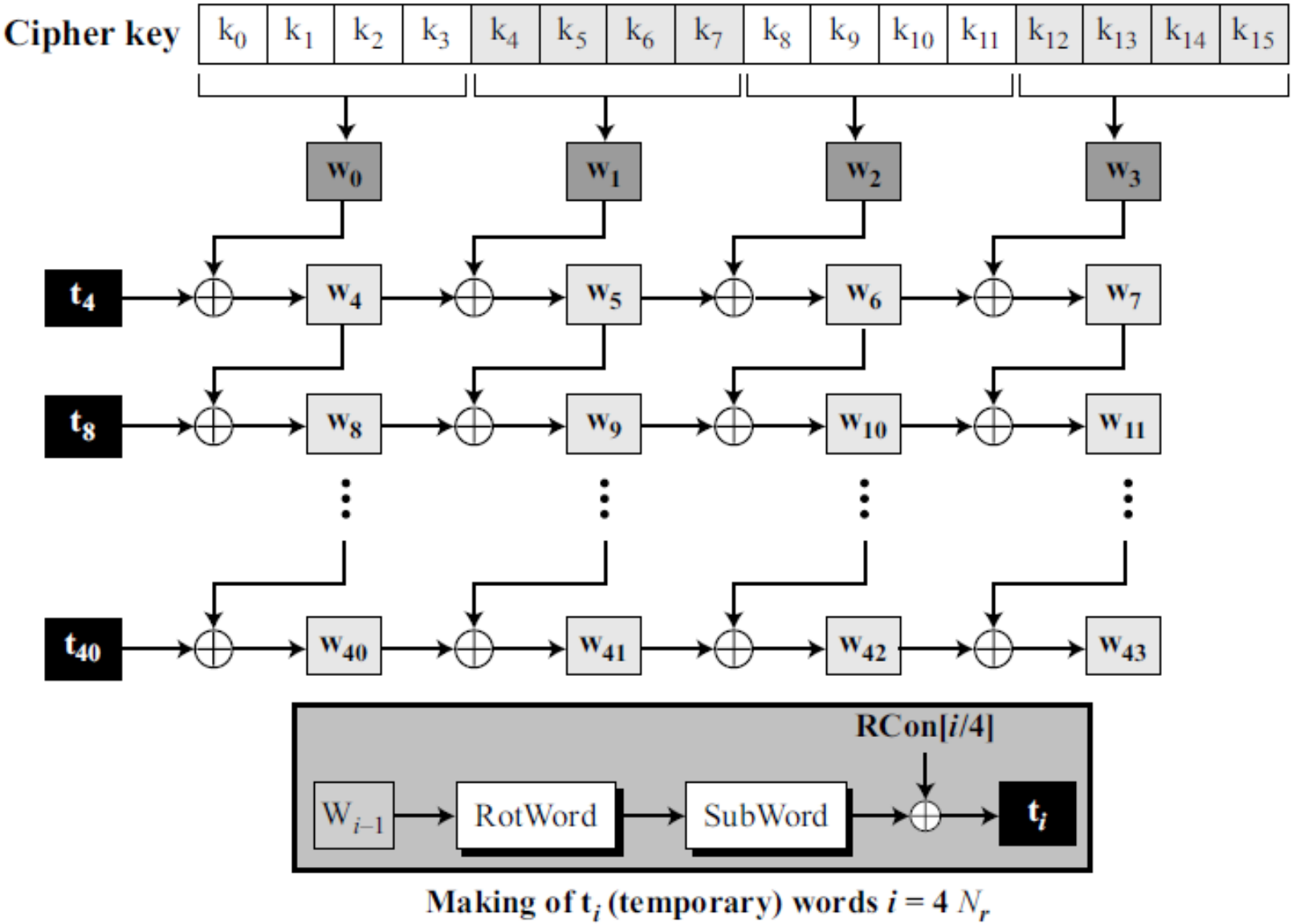
MixColumns transformation



AddRoundKey transformation



Key expansion in AES



Making of t_i (temporary) words $i = 4\ N_r$

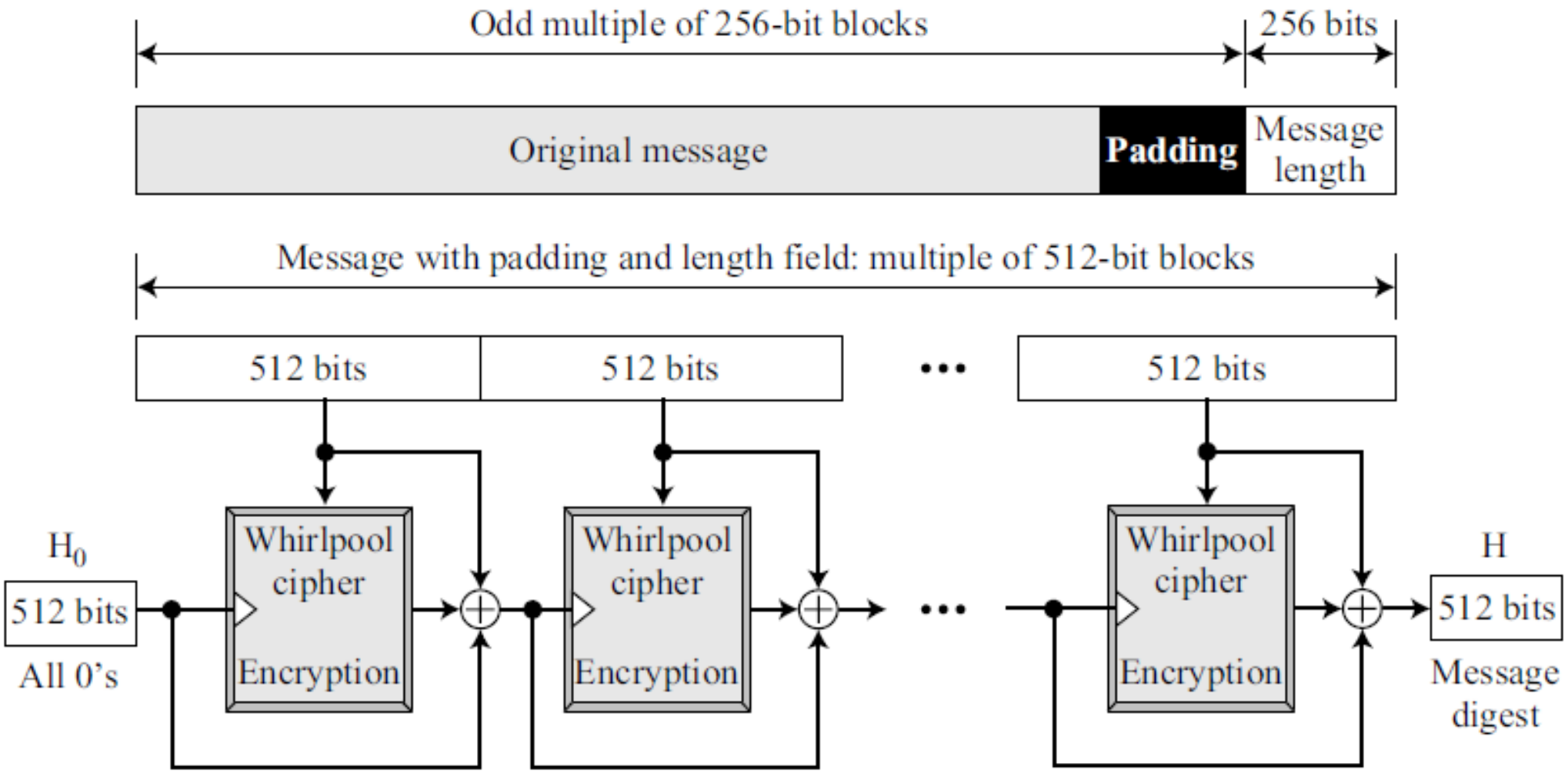
Round Constants

Round	*Constant (RCon)*	*Round*	*Constant (RCon)*
1	$(\mathbf{01}\ 00\ 00\ 00)_{16}$	6	$(\mathbf{20}\ 00\ 00\ 00)_{16}$
2	$(\mathbf{02}\ 00\ 00\ 00)_{16}$	7	$(\mathbf{40}\ 00\ 00\ 00)_{16}$
3	$(\mathbf{04}\ 00\ 00\ 00)_{16}$	8	$(\mathbf{80}\ 00\ 00\ 00)_{16}$
4	$(\mathbf{08}\ 00\ 00\ 00)_{16}$	9	$(\mathbf{1B}\ 00\ 00\ 00)_{16}$
5	$(\mathbf{10}\ 00\ 00\ 00)_{16}$	10	$(\mathbf{36}\ 00\ 00\ 00)_{16}$

Whirlpool

- Iterated cryptographic hash function, based on the Miyaguchi-Preneel scheme, that uses a symmetric-key block cipher in place of the compression function.
- The block cipher is a modified AES cipher that has been tailored for this purpose
- the augmented message size is an even multiple of 256 bits or a multiple of 512 bits.
- Whirlpool creates a digest of 512 bits from a multiple 512-bit block message
- The 512-bit digest, H0, is initialized to all 0’s.
- This value becomes the cipher key for encrypting the first block
- The ciphertext resulting from encrypting each block becomes the cipher key for the next block after being exclusive-ored with the previous cipher key and the plaintext block.
- The message digest is the final 512-bit ciphertext after the last exclusive-or operation.

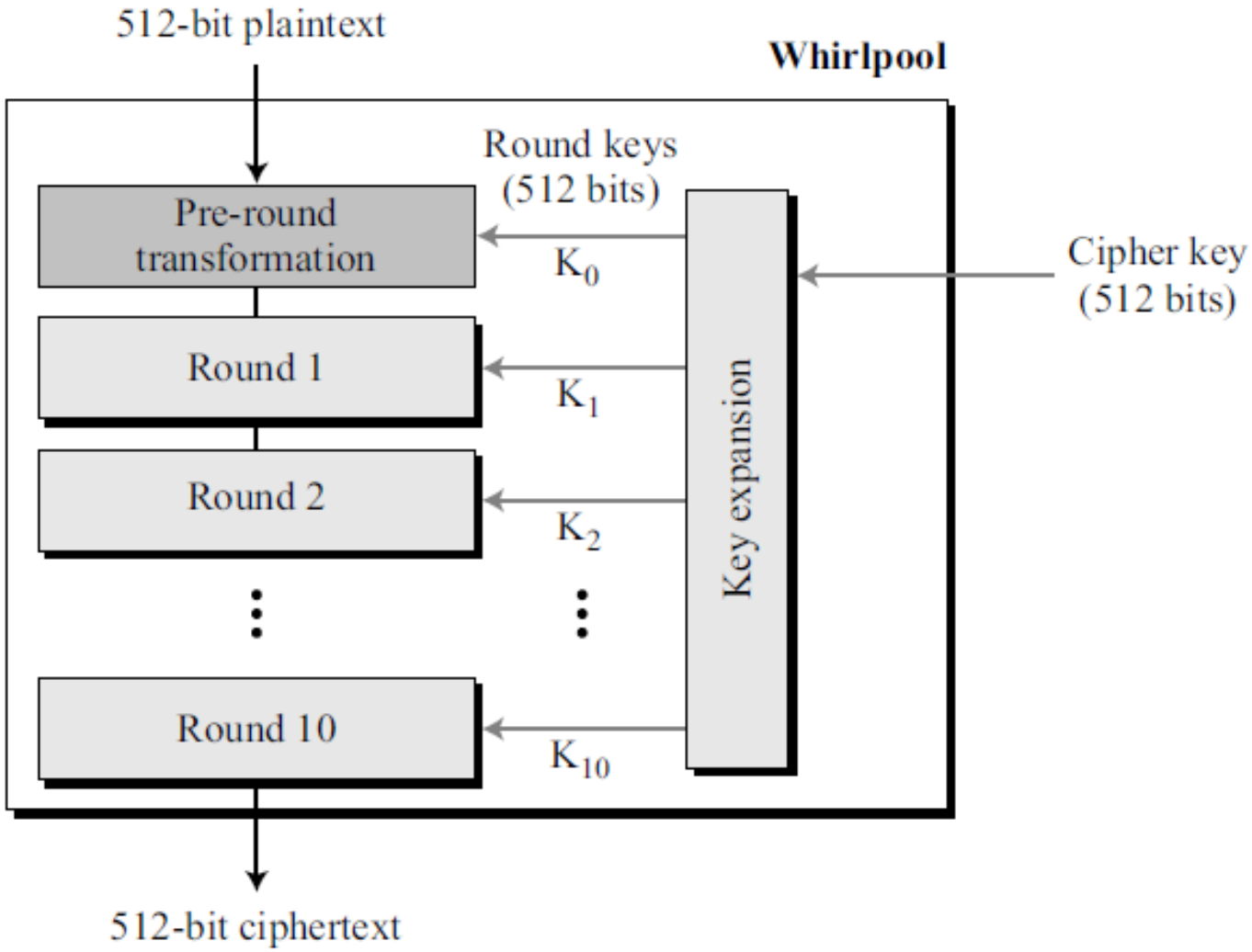
Whirlpool hash function



Whirlpool Cipher

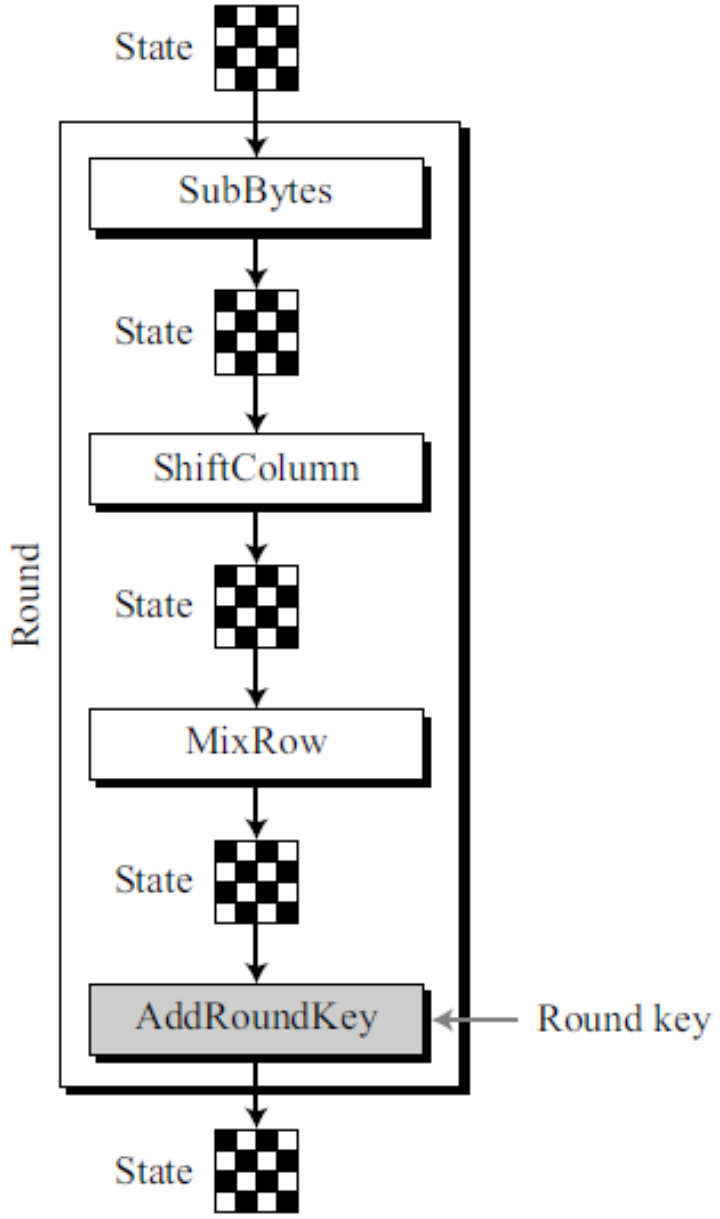
- The Whirlpool cipher is a non-Feistel cipher like AES that was mainly designed as a block cipher to be used in a hash algorithm
- ***Rounds***
- Whirlpool is a round cipher that uses 10 rounds
- The block size and key size are 512 bits
- The cipher uses 11 round keys, K0 to K10, each of 512 bits

General idea of the Whirlpool cipher

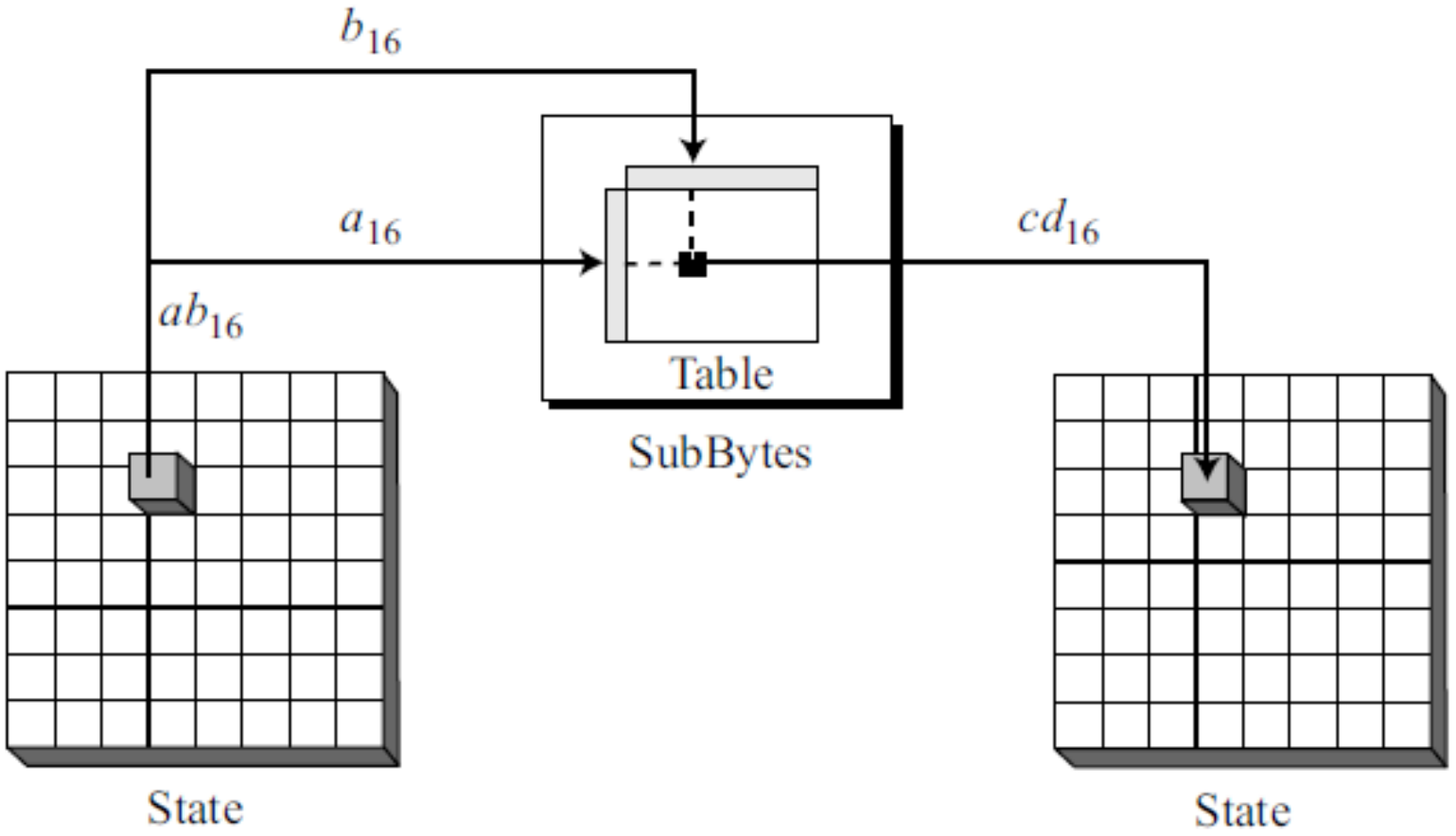


- size of the block or state is 512 bits
- A block is considered as a row matrix of **64 bytes**; a state is considered as a square matrix of **8 × 8 bytes**
- the block-to-state or state -to-block transformation is done **row by row**

Structure of each round in the Whirlpool cipher



SubBytes transformations in the Whirlpool cipher

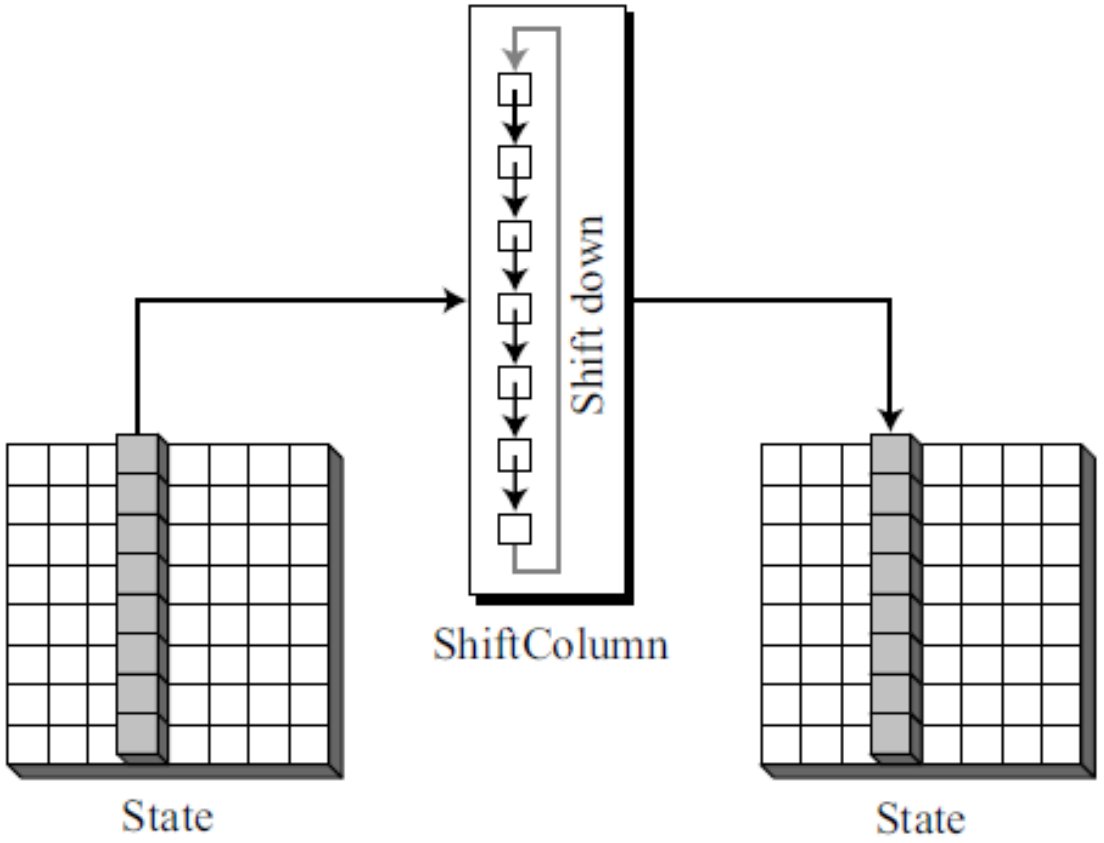


SubBytes transformation table (S-Box)

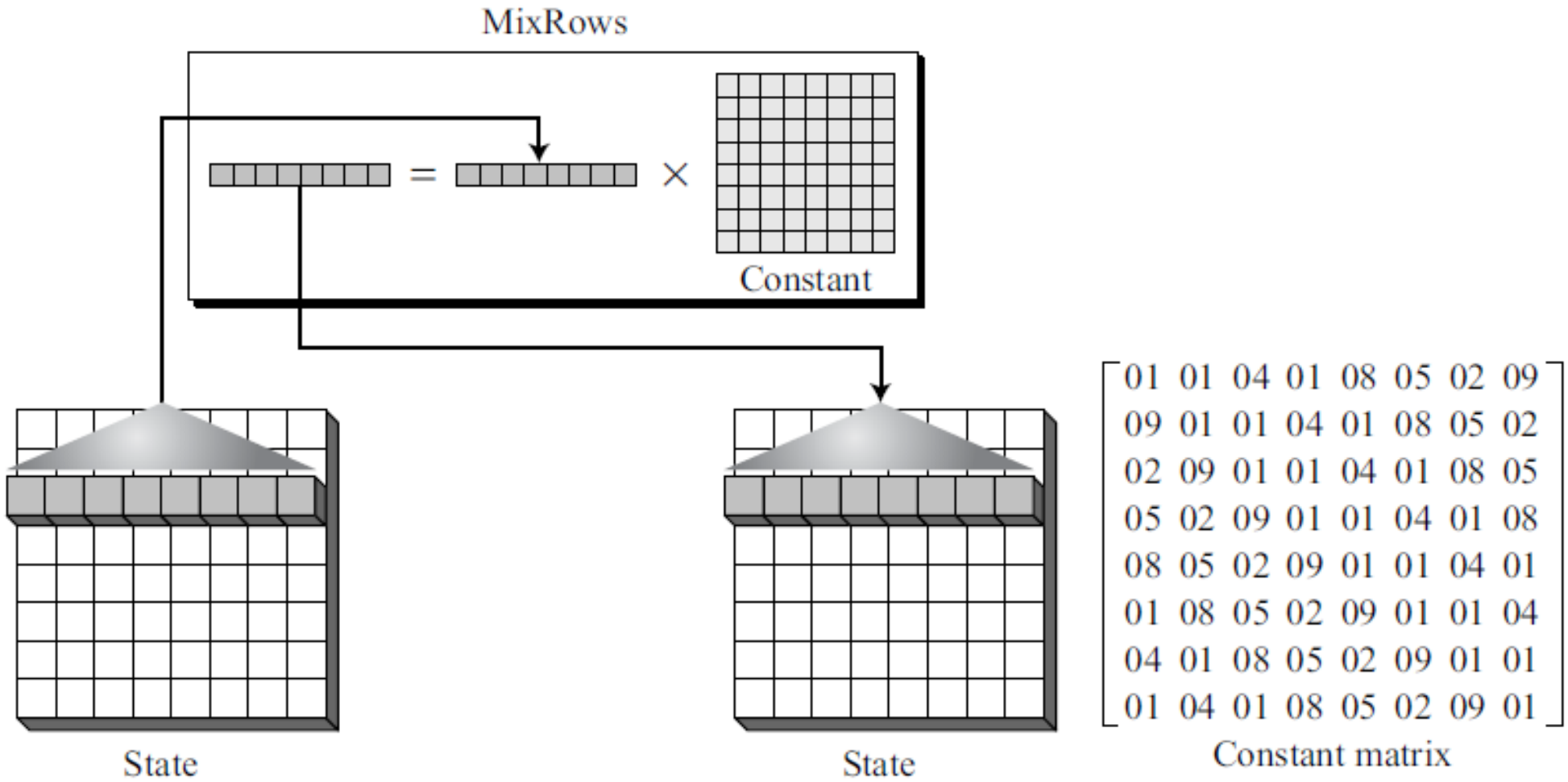
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*A*	*B*	*C*	*D*	*E*	*F*
0	18	23	C6	E8	87	B8	01	4F	36	A6	D2	F5	79	6F	91	52
1	16	BC	9B	8E	A3	0C	7B	35	1D	E0	D7	C2	2E	4B	FE	57
2	15	77	37	E5	9F	F0	4A	CA	58	C9	29	0A	B1	A0	6B	85
3	BD	5D	10	F4	CB	3E	05	67	E4	27	41	8B	A7	7D	95	C8
4	FB	EF	7C	66	DD	17	47	9E	CA	2D	BF	07	AD	5A	83	33
5	63	02	AA	71	C8	19	49	C9	F2	E3	5B	88	9A	26	32	B0
6	E9	0F	D5	80	BE	CD	34	48	FF	7A	90	5F	20	68	1A	AE
7	B4	54	93	22	64	F1	73	12	40	08	C3	EC	DB	A1	8D	3D
8	97	00	CF	2B	76	82	D6	1B	B5	AF	6A	50	45	F3	30	EF
9	3F	55	A2	EA	65	BA	2F	C0	DE	1C	FD	4D	92	75	06	8A
A	B2	E6	0E	1F	62	D4	A8	96	F9	C5	25	59	84	72	39	4C
B	5E	78	38	8C	C1	A5	E2	61	B3	21	9C	1E	43	C7	FC	04
C	51	99	6D	0D	FA	DF	7E	24	3B	AB	CE	11	8F	4E	B7	EB
D	3C	81	94	F7	9B	13	2C	D3	E7	6E	C4	03	56	44	7E	A9
E	2A	BB	C1	53	DC	0B	9D	6C	31	74	F6	46	AC	89	14	E1
F	16	3A	69	09	70	B6	C0	ED	CC	42	98	A4	28	5C	F8	86

ShiftColumns

- similar to the ShiftRows transformation in AES, except that the columns instead of rows are shifted.
- Shifting depends on the position of the column
- Column 0 goes through 0-byte shifting (no shifting), while column 7 goes through 7-byte shifting.



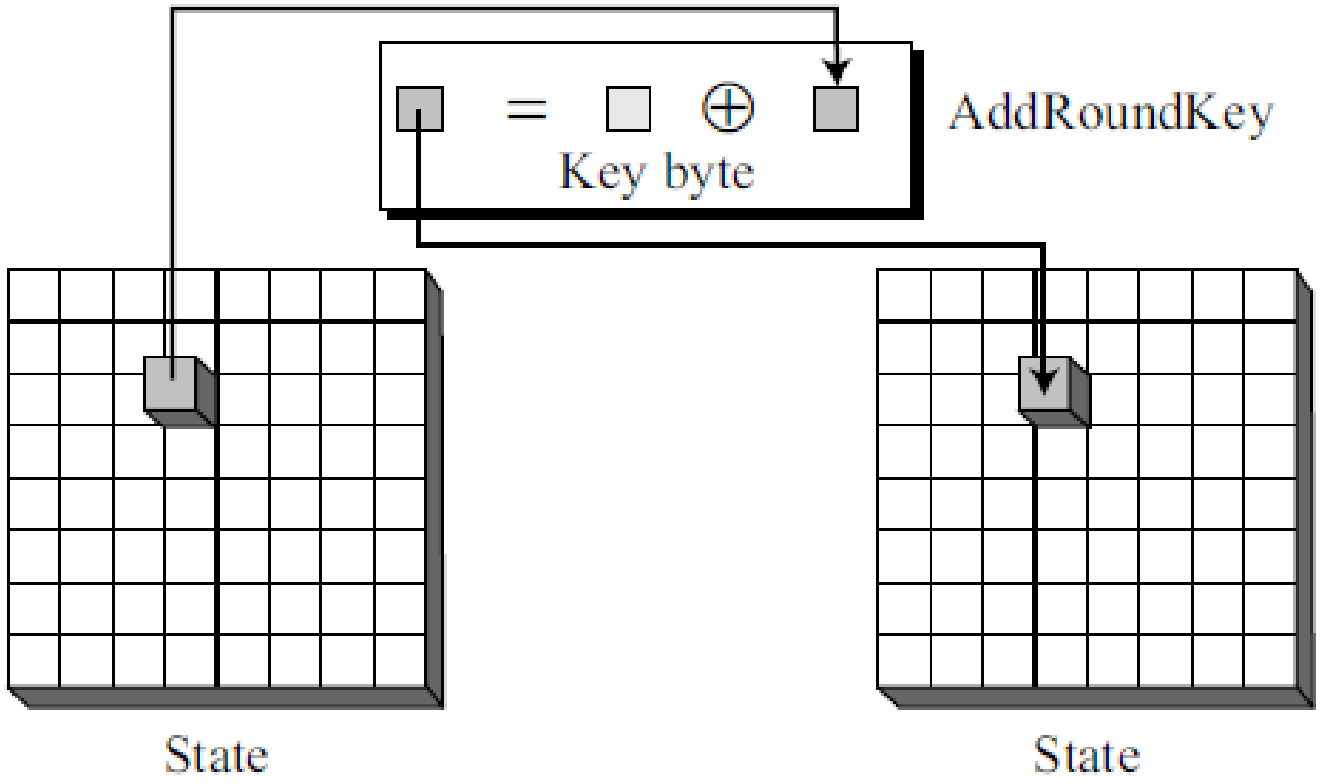
MixRows transformation in the Whirlpool cipher



$$\begin{bmatrix} 01 & 01 & 04 & 01 & 08 & 05 & 02 & 09 \\ 09 & 01 & 01 & 04 & 01 & 08 & 05 & 02 \\ 02 & 09 & 01 & 01 & 04 & 01 & 08 & 05 \\ 05 & 02 & 09 & 01 & 01 & 04 & 01 & 08 \\ 08 & 05 & 02 & 09 & 01 & 01 & 04 & 01 \\ 01 & 08 & 05 & 02 & 09 & 01 & 01 & 04 \\ 04 & 01 & 08 & 05 & 02 & 09 & 01 & 01 \\ 01 & 04 & 01 & 08 & 05 & 02 & 09 & 01 \end{bmatrix}$$

Constant matrix

AddRoundKey transformation in the Whirlpool cipher



Key Expansion

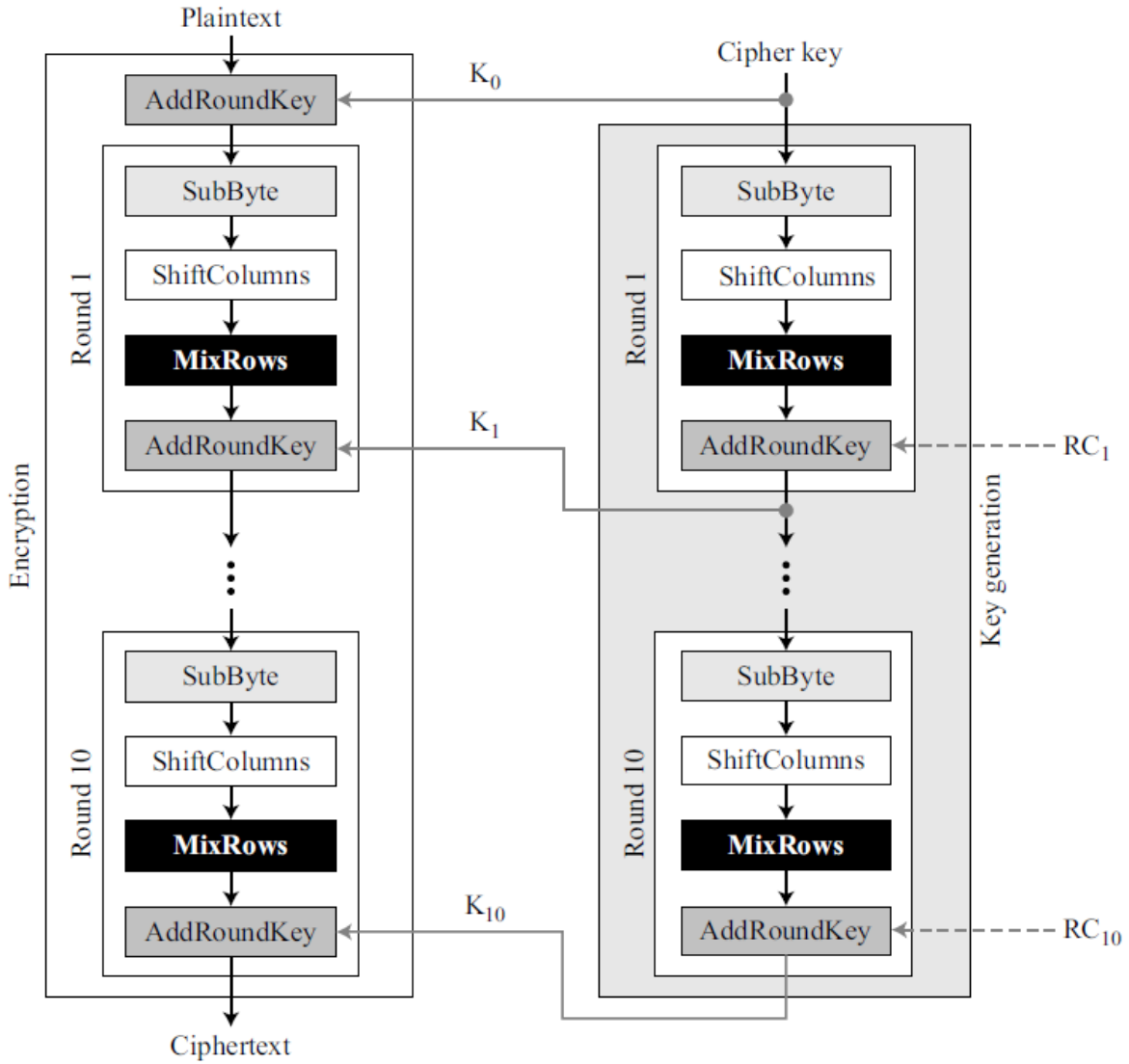
- Instead of using a new algorithm for creating round keys, Whirlpool uses a copy of the encryption algorithm (without the pre-round) to create the round keys.
- The output of each round in the encryption algorithm is the round key for that round
- key-expansion algorithm uses constants as the round keys and the encryption algorithm uses the output of each round of the key expansion algorithm as the round keys.
- The key-generation algorithm treats the cipher key as the plaintext and encrypts it.
- Note that the cipher key is also K0 for theencryption algorithm.
- Round Constants Each round

Round Constants

- Each round constant, RCr is an 8 × 8 matrix where only the first row has non-zero values.
- The rest of the entries are all 0's.
- The values for the first row in each constant matrix can be calculated using the SubBytes transformation
- RC1 uses the first eight entries in the SubBytes transformation table, RC2 uses the second eight entries, and so on.

$$RC_{round}[row, column] = SubBytes\,(8(round-1)+column) \qquad \text{if } row = 0$$
$$RC_{round}\,[row, column] = 0 \qquad \text{if } row \neq 0$$

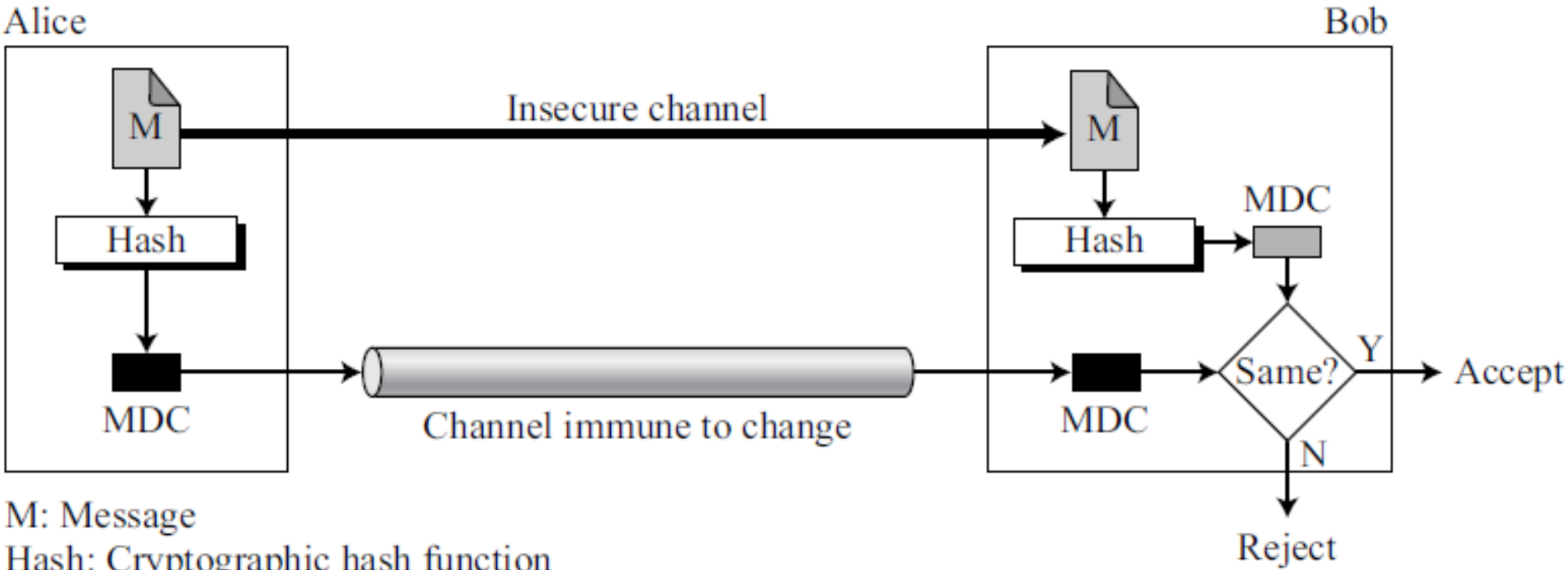
Key expansion in the Whirlpool cipher



MESSAGE AUTHENTICATION

- A message digest guarantees the integrity of a message
- It guarantees that the message has not been changed
- A message digest does not authenticate the sender of the message.
- When Alice sends a message to Bob, Bob needs to know if the message is coming from Alice.
- To provide message authentication, Alice needs to provide proof that it is Alice sending the message and not an impostor.
- A message digest cannot provide such a proof.
- The digest created by a cryptographic hash function is normally called a modification detection code (MDC).
- The code can detect any modification in the message.
- What we need for message authentication (data origin authentication) is a message authentication code (MAC).

Modification detection code (MDC)



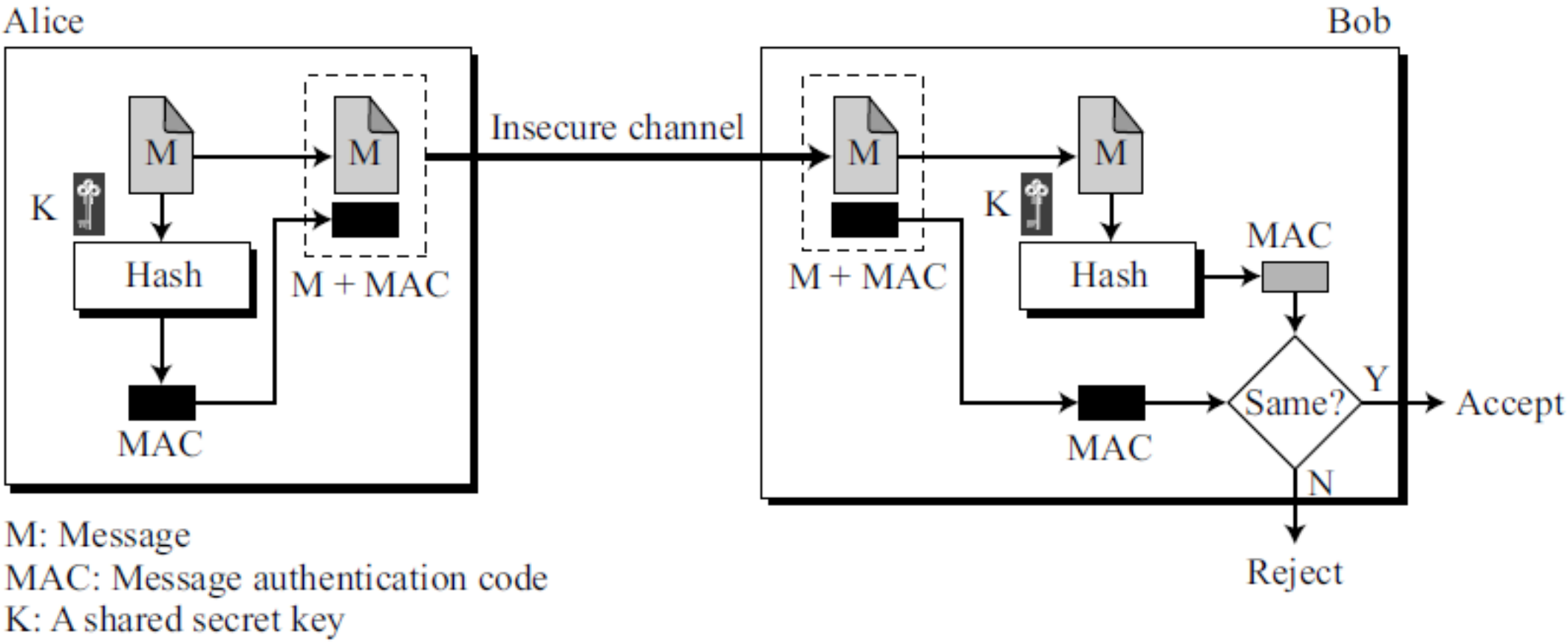
Modification Detection Code

- is a message digest that can prove the integrity of the message
- If Alice needs to send a message to Bob and be sure that the message will not change during transmission, Alice can create a message digest, MDC, and send both the message and the MDC to Bob.
- Bob can create a new MDC from the message and compare the received MDC and the new MDC. If they are the same, the message has not been changed.
- Note: the message can be transferred through an insecure channel.
- Eve can read or even modify the message.
- The MDC, however, needs to be transferred through a safe channel.
- The term safe here means immune to change.
- If both the message and the MDC are sent through the insecure channel, Eve can intercept the message, change it, create a new MDC from the message, and send both to Bob.
- Bob never knows that the message has come from Eve.

Message Authentication Code (MAC)

- To ensure the integrity of the message and the data origin authentication
- MAC includes a secret between Alice and Bob for example, a secret key that Eve does not possess.
- Alice uses a hash function to create a MAC from the concatenation of the key and the message, h (K|M).
- She sends the message and the MAC to Bob over the insecure channel.
- Bob separates the message from the MAC.
- He then makes a new MAC from the concatenation of the message and the secret key.
- Bob then compares the newly created MAC with the one received.
- If the two MACs match, the message is authentic and has not been modified by an adversary.

Message authentication code



If only the receiver and the sender know the identity of the secret key, and if the received MAC matches the calculated MAC, then

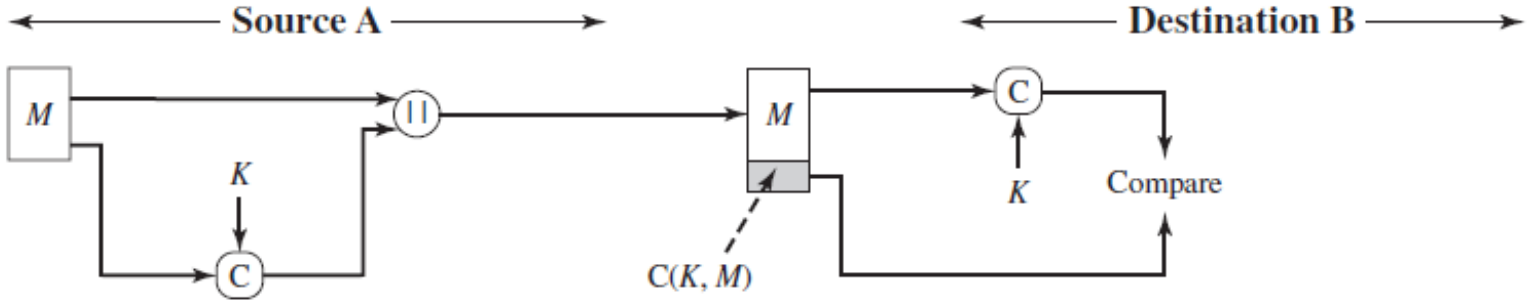
1. The receiver is assured that the message has not been altered.
   - If an attacker alters the message but does not alter the MAC, then the receiver's calculation of the MAC will differ from the received MAC. Because the attacker is assumed not to know the secret key, the attacker cannot alter the MAC to correspond to the alterations in the message.

2. The receiver is assured that the message is from the alleged sender.

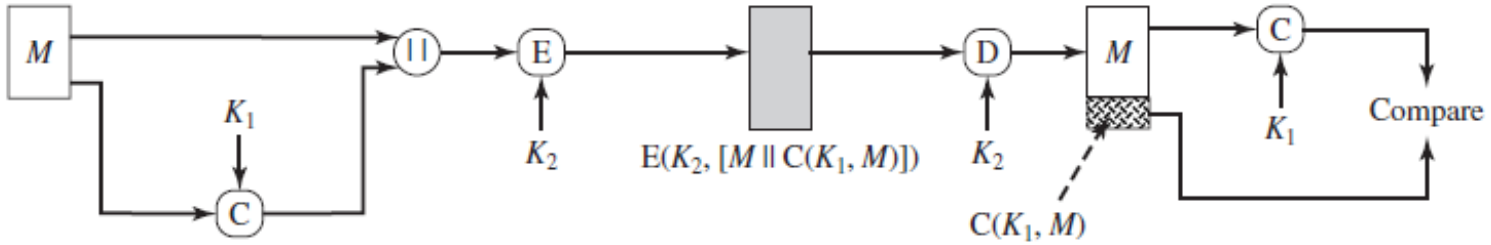
- Because no one else knows the secret key, no one else could prepare a message with a proper MAC.

3. If the message includes a sequence number, then the receiver can be assured of the proper sequence because an attacker cannot successfully alter the sequence number.

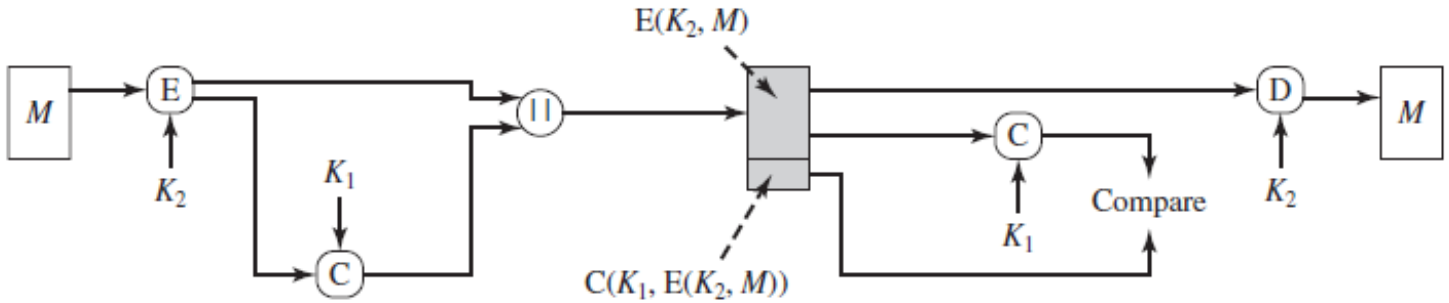
Basic Uses of Message Authentication code(MAC)



(a) Message authentication



(b) Message authentication and confidentiality; authentication tied to plaintext



(c) Message authentication and confidentiality; authentication tied to ciphertext

- Fig a provides authentication but not confidentiality, because the message as a whole is transmitted in the clear.
- Confidentiality can be provided by performing message encryption either after (Figure b) or before (Figure c) the MAC algorithm.
- In both these cases, two separate keys are needed, each of which is shared by the sender and the receiver.
- In the first case, the MAC is calculated with the message as input and is then concatenated to the message. The entire block is then encrypted.
- In the second case, the message is encrypted first. Then the MAC is calculated using the resulting ciphertext and is concatenated to the ciphertext to form the transmitted block.
- Typically, it is preferable to tie the authentication directly to the plaintext, so the method of Figure b is used.

Situations in which a message authentication code is used.

Symmetric encryption will provide authentication and because it is widely used with readily available products, why not simply use this instead of a separate message authentication code?

1. There are a number of applications in which the same message is broadcast to a number of destinations.
   - It is cheaper and more reliable to have only one destination responsible for monitoring authenticity. Thus, the message must be broadcast in plaintext with an associated message authentication code. The responsible system has the secret key and performs authentication. If a violation occurs, the other destination systems are alerted by a general alarm.

2. Another possible scenario is an exchange in which one side has a heavy load and cannot afford the time to decrypt all incoming messages.

- Authentication is carried out on a selective basis, messages being chosen at random for checking.

3. Authentication of a computer program in plaintext is an attractive service.

- The computer program can be executed without having to decrypt it every time, which would be wasteful of processor resources. However, if a message authentication code were attached to the program, it could be checked whenever assurance was required of the integrity of the program.

4. For some applications, it may not be of concern to keep messages secret, but it is important to authenticate messages.

5. Separation of authentication and confidentiality functions affords architectural flexibility. For example, it may be desired to perform authentication at the application level but to provide confidentiality at a lower level, such as the transport layer

6. A user may wish to prolong the period of protection beyond the time of reception and yet allow processing of message contents

- With message encryption, the protection is lost when the message is decrypted, so the message is protected against fraudulent modifications only in transit but not within the target system.

- The MAC is referred to as a prefix MAC because the secret key is appended to the beginning of the message.
- postfix MAC, in which the key is appended to the end of the message.
- can combine the prefix and postfix MAC, with the same key or two different keys. However, the resulting MACs are still insecure.

Security of a MAC

- Suppose Eve has intercepted the message M and the digest h(K|M).
- How can Eve forge a message without knowing the secret key?
- There are three possible cases:

1. If the size of the key allows exhaustive search,
   - prepend all possible keys at the beginning of the message and make a digest of the (K|M) to find the digest equal to the one intercepted.
   - replace the message with a forged message

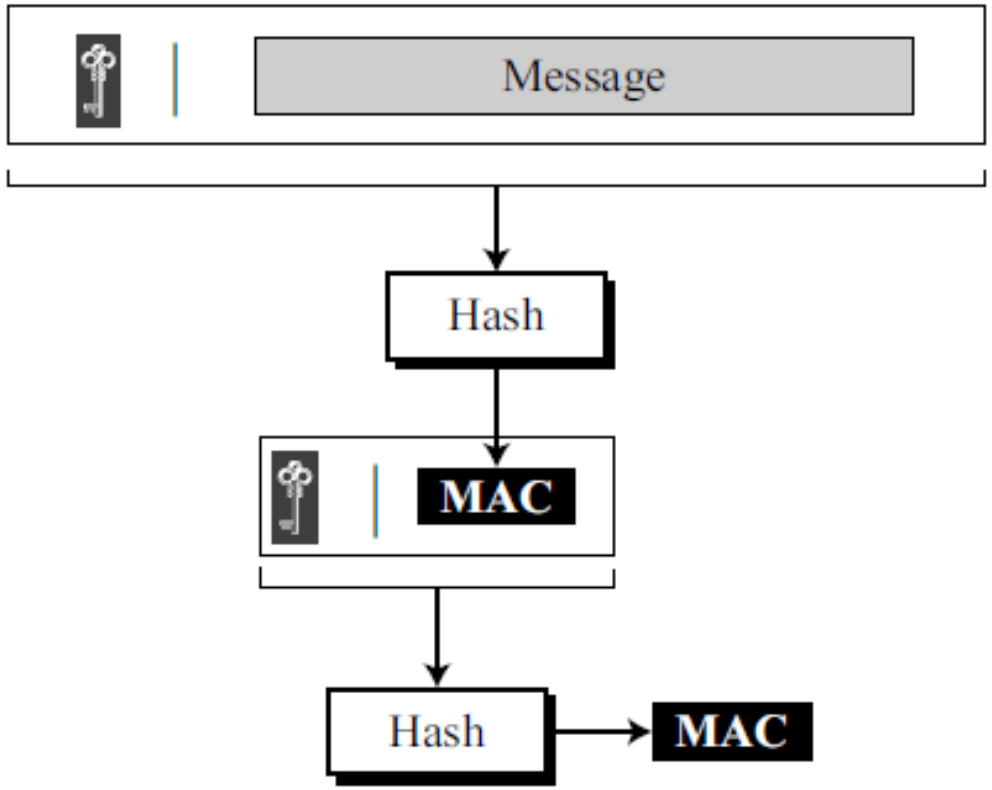
2. The size of the key is normally very large in a MAC use preimage attack
   - use the algorithm , find X such that h(X) is equal to the MAC and find the key
   - successfully replace the message with a forged one.

3. Given some pairs of messages and their MACs, Eve can manipulate them to come up with a new message and its MAC.

The security of a MAC depends on the security of the underlying hash algorithm.

Nested MAC

- hashing is done in two steps.
- In the first step, the key is concatenated with the message and is hashed to create an intermediate digest.
- In the second step, the key is concatenated with the intermediate digest to create the final digest.



REQUIREMENTS FOR MESSAGE AUTHENTICATION CODES

1. If an opponent observes M and $\mathrm{MAC}(K, M)$, it should be computationally infeasible for the opponent to construct a message M' such that

$$\mathrm{MAC}(K, M') = \mathrm{MAC}(K, M)$$

2. $\mathrm{MAC}(K, M)$ should be uniformly distributed in the sense that for randomly chosen messages, M and M', the probability that $\mathrm{MAC}(K, M) = \mathrm{MAC}(K, M')$ is 2^{-n}, where n is the number of bits in the tag.
3. Let M' be equal to some known transformation on M. That is, $M' = \mathrm{f}(M)$. For example, f may involve inverting one or more specific bits. In that case,

$$\Pr[\mathrm{MAC}(K, M) = \mathrm{MAC}(K, M')] = 2^{-n}$$

HMAC(hashed MAC) -Steps

1. The message is divided into N blocks, each of b bits.

2. The secret key is left-padded with 0’s to create a b-bit key.

The secret key (before padding) be longer than n bits, where n is the size of the HMAC.

3. The result of step 2 is exclusive-ored with a constant called ipad (input pad) to create a b-bit block. The value of ipad is the b/8 repetition of the sequence 00110110 (36 in hexadecimal).

4. The resulting block is prepended to the N-block message. The result is N + 1 blocks.

5. The result of step 4 is hashed to create an n-bit digest. We call the digest the intermediate HMAC.

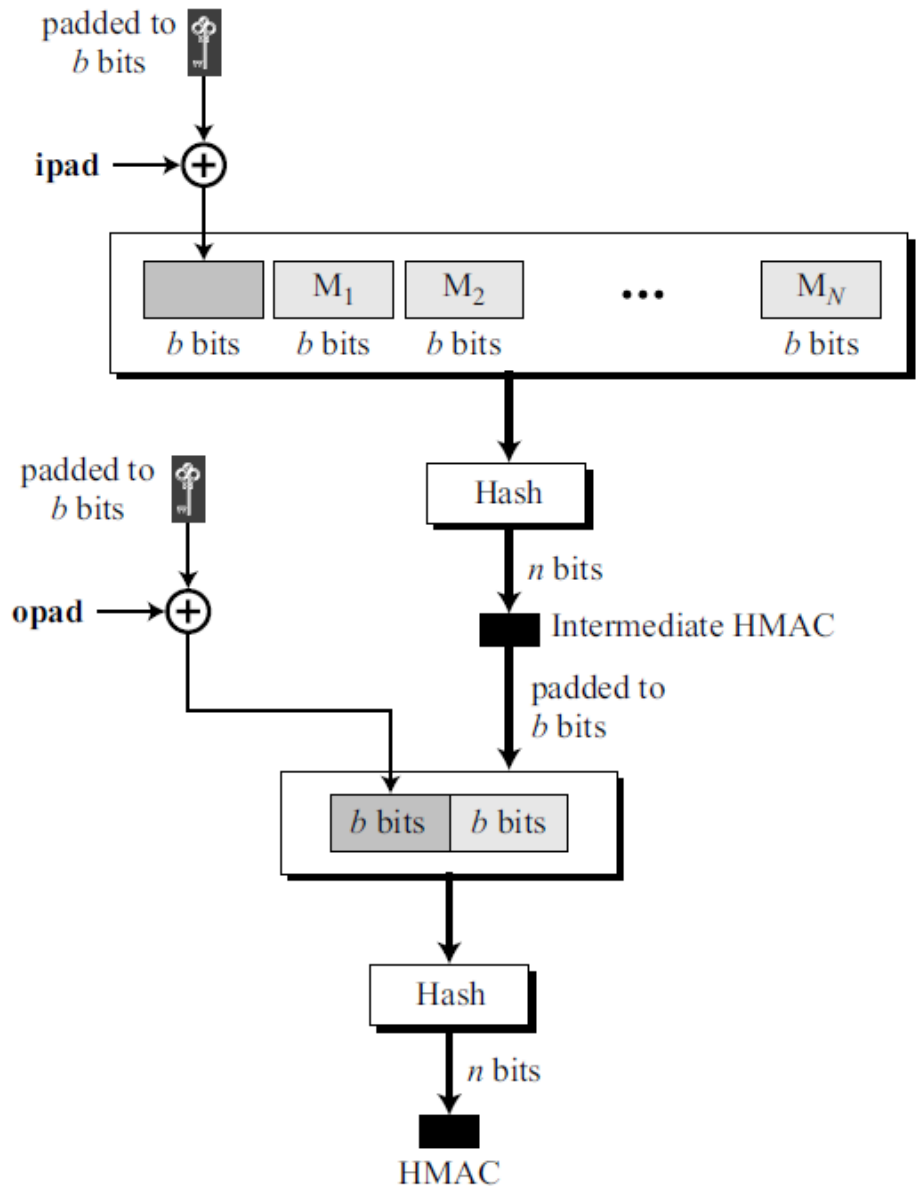
HMAC(hashed MAC) –Steps Contd…

6. The intermediate n-bit HMAC is left padded with 0s to make a b-bit block.

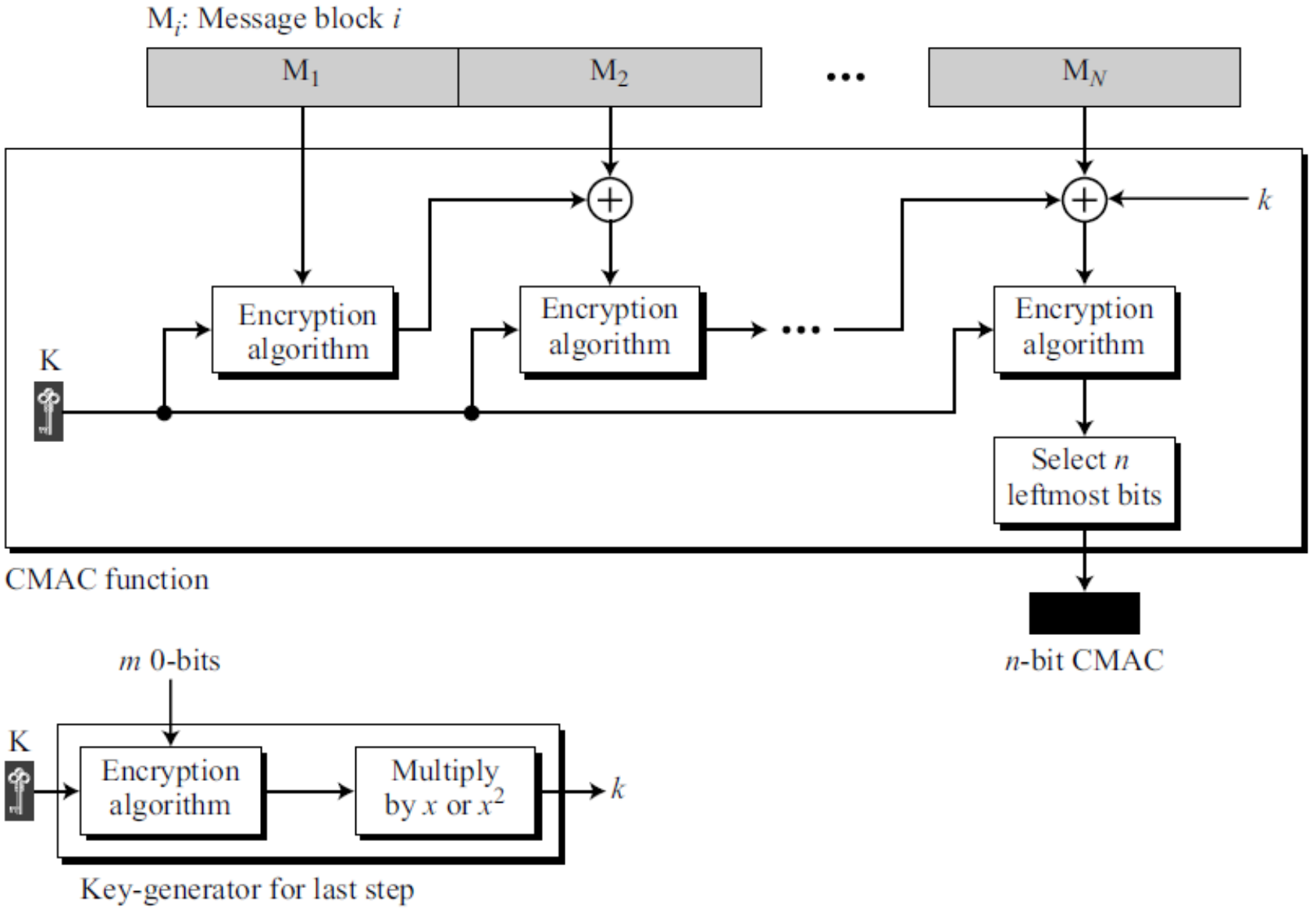
7. Steps 2 and 3 are repeated by a different constant opad (output pad). The value of opad is the b/8 repetition of the sequence 01011100 (5C in hexadecimal).

8. The result of step 7 is prepended to the block of step 6.

9. The result of step 8 is hashed with the same hashing algorithm to create the final n-bit HMAC.


padded to
b bits
ipad
M1
M2
M_N
b bits
b bits
b bits
b bits
padded to
b bits
opad
Hash
n bits
Intermediate HMAC
padded to
b bits
b bits
b bits
Hash
n bits
HMAC

CMAC called Data Authentication Algorithm, or CMAC, or CBCMAC



Steps

- The message is divided into N blocks, each m bits long
- The size of the CMAC is n bits
- If the last block is not m bits, it is padded with a 1-bit followed by enough 0-bits to make it m bits.
- The first block of the message is encrypted with the symmetric key to create an m-bit block of encrypted data
- This block is XORed with the next block and the result is encrypted again to create a new m-bit block

Steps

- The process continues until the last block of the message is encrypted
- The n leftmost bit from the last block is the CMAC.
- In addition to the symmetric key, K, CMAC also uses another key, k, which is applied only at the last step.
  - This key is derived from the encryption algorithm with plaintext of m 0-bits using the cipher key, K. The result is then multiplied by x if no padding is applied and multiplied by x^2 if padding is applied.
  - The multiplication is in GF(2^m) with the irreducible polynomial of degree m selected by the particular protocol used.

References

- Behrouz A. Forouzan and Debdeep Mukhopadhyay – “Cryptography and Network Security”, McGraw Hill, 2nd Edition, 2008.
- William Stallings,“Cryptography And Network Security Principles And Practice”, Fifth Edition, Pearson Education, 2013