MODULE 4

Message Integrity and Message Authentication

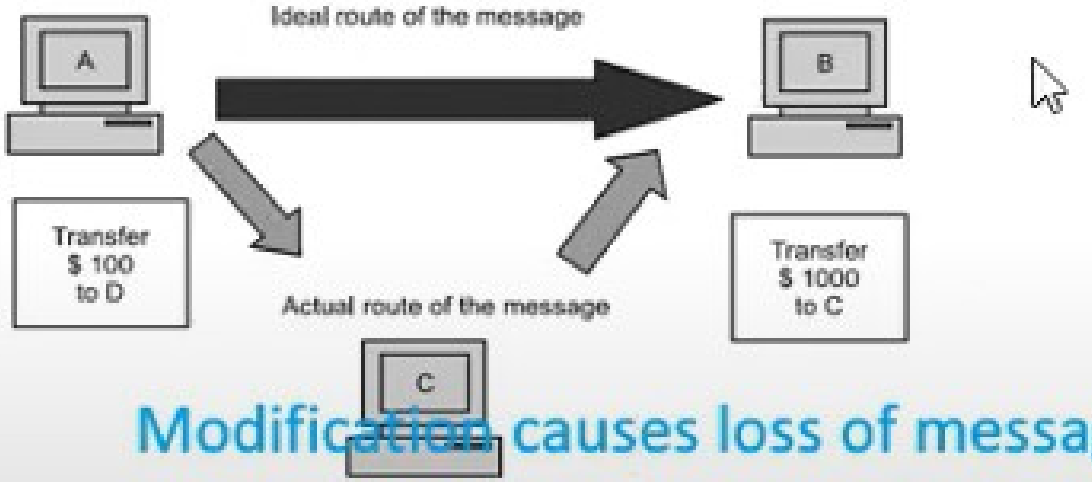
MESSAGE INTEGRITY

Cryptography systems that discussed so far provide secrecy, or confidentiality, but not integrity.

There are occasions where we may not even need secrecy but instead must have integrity.

MESSAGE INTEGRITY

Here, user C tampers with a message originally sent by user A, which is actually destined for user B. User C somehow manages to access it, change its contents, and send the changed message to user B. User B has no way of knowing that the contents of the message were changed after user A had sent it. User A also does not know about this change. This type of attack is called **modification**.



Modification causes loss of message integrity.

MESSAGE INTEGRITY

- Alice may write a will to distribute her estate upon her death.
- The will does not need to be encrypted.
- After her death, anyone can examine the will.
- The integrity of the will, however, needs to be preserved.
  - ✓ Alice does not want the contents of the will to be changed.

Document and Fingerprint

One way to preserve the integrity of a document is through the use of a fingerprint.

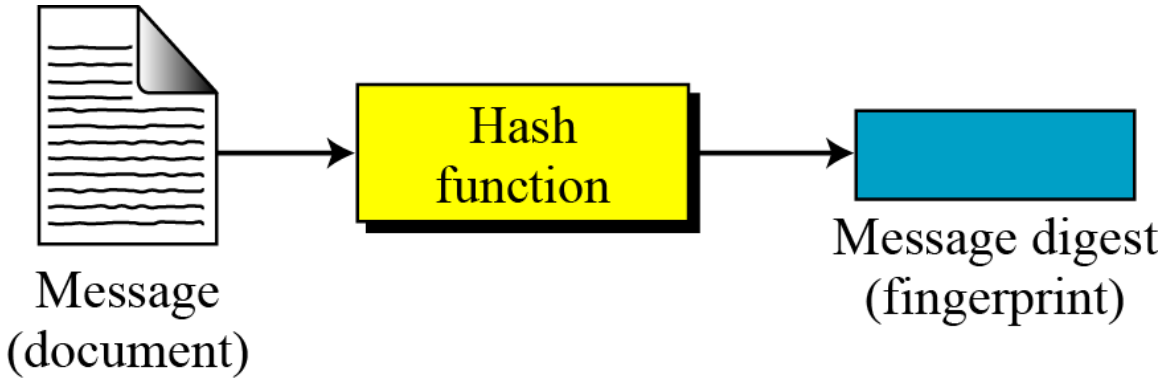
- Alice can put her fingerprint at the bottom of the document.
- Eve cannot modify the contents of this document or create a false document because she cannot forge Alice's fingerprint.
- To ensure that the document has not been changed, Alice's fingerprint on the document can be compared to Alice's 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.

Figure 11.1 *Message and digest*



Difference

Two pairs (document/fingerprint) and (message/message digest) are similar, with some differences.

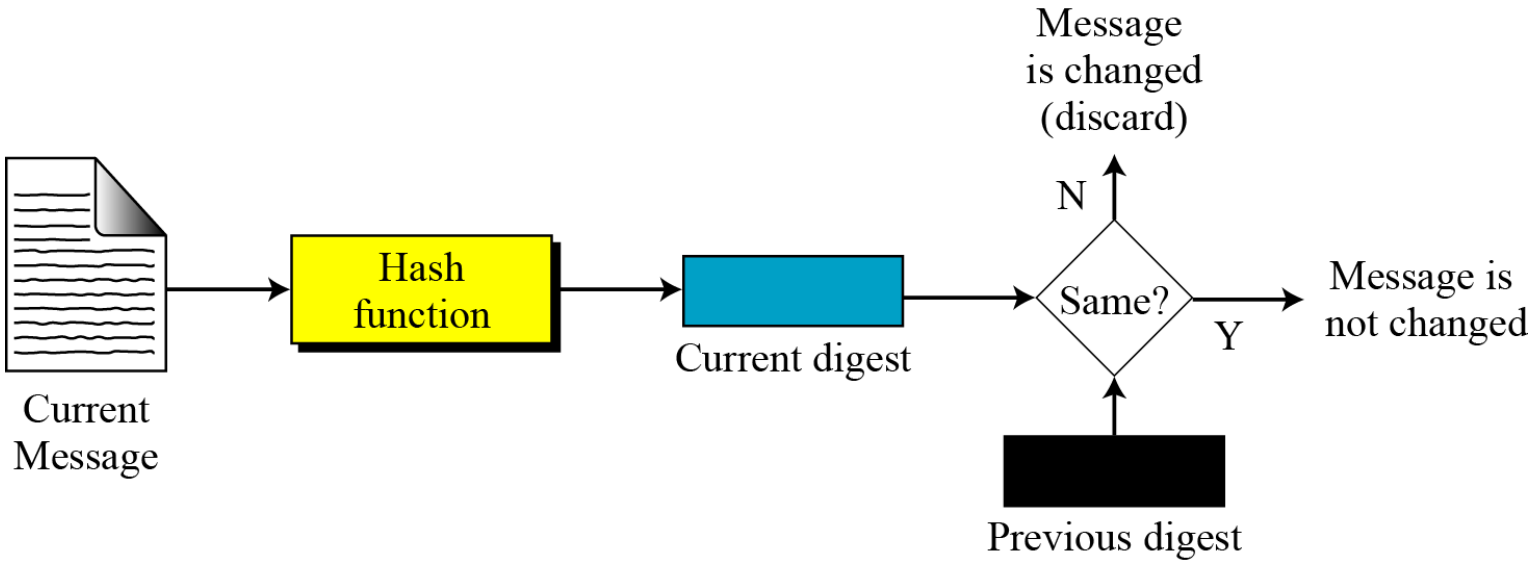
- Document and fingerprint are physically linked together.
- Message and message digest can be unlinked (or sent) separately, and, most importantly, the message digest needs to be safe from change.

The message digest needs to be safe from change.

Checking Integrity

Run the cryptographic hash function again and compare the new message digest with the previous one.

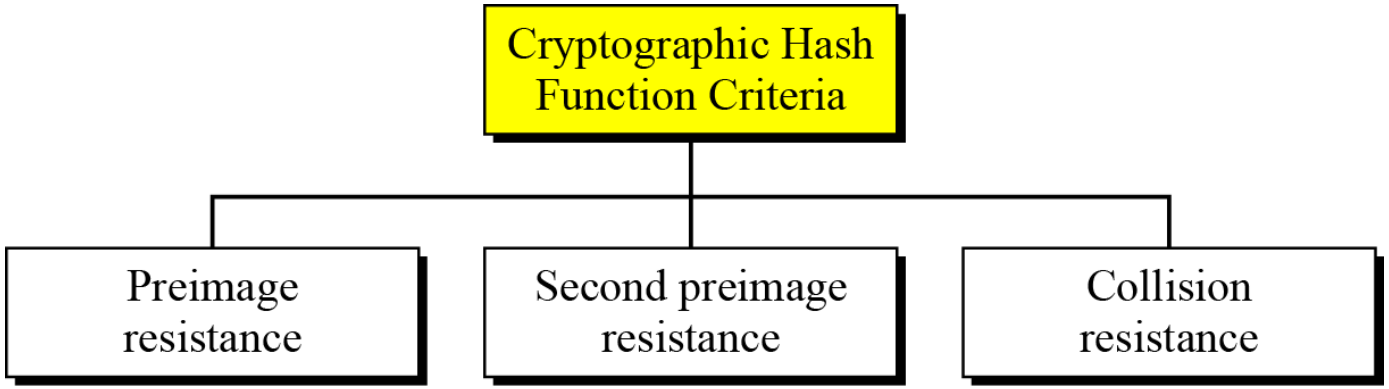
Figure 11.2 *Checking integrity*



Cryptographic Hash Function Criteria

A cryptographic hash function must satisfy three criteria:

Figure 11.3 **Criteria of a cryptographic hash function**

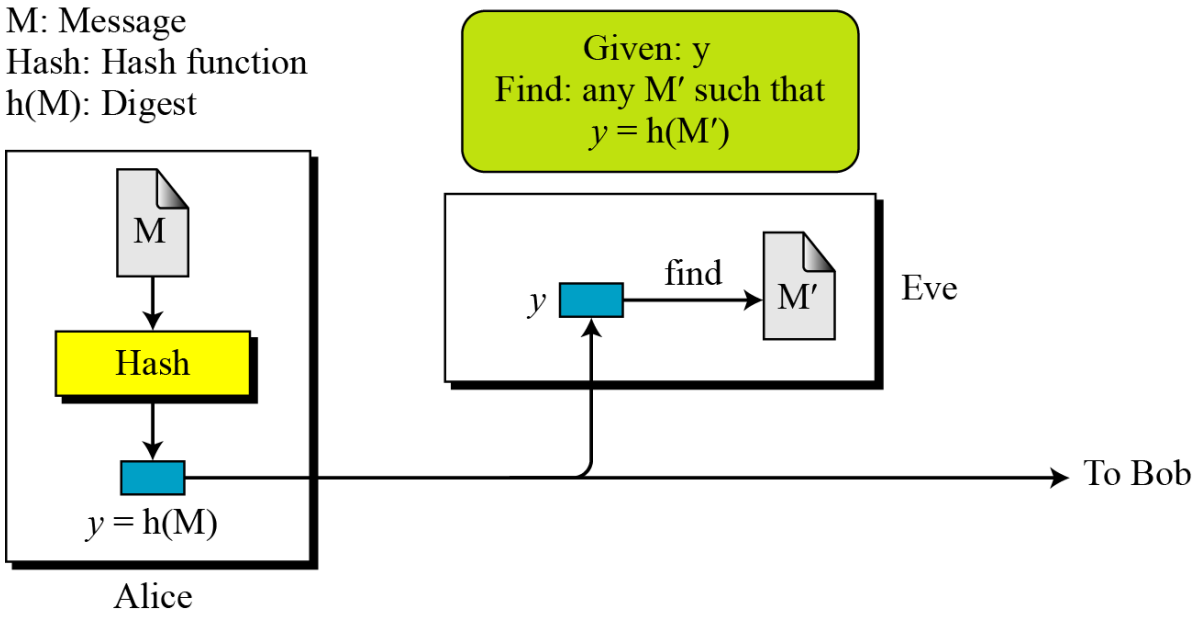


Preimage Resistance

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′).

Figure 11.4 *Preimage*



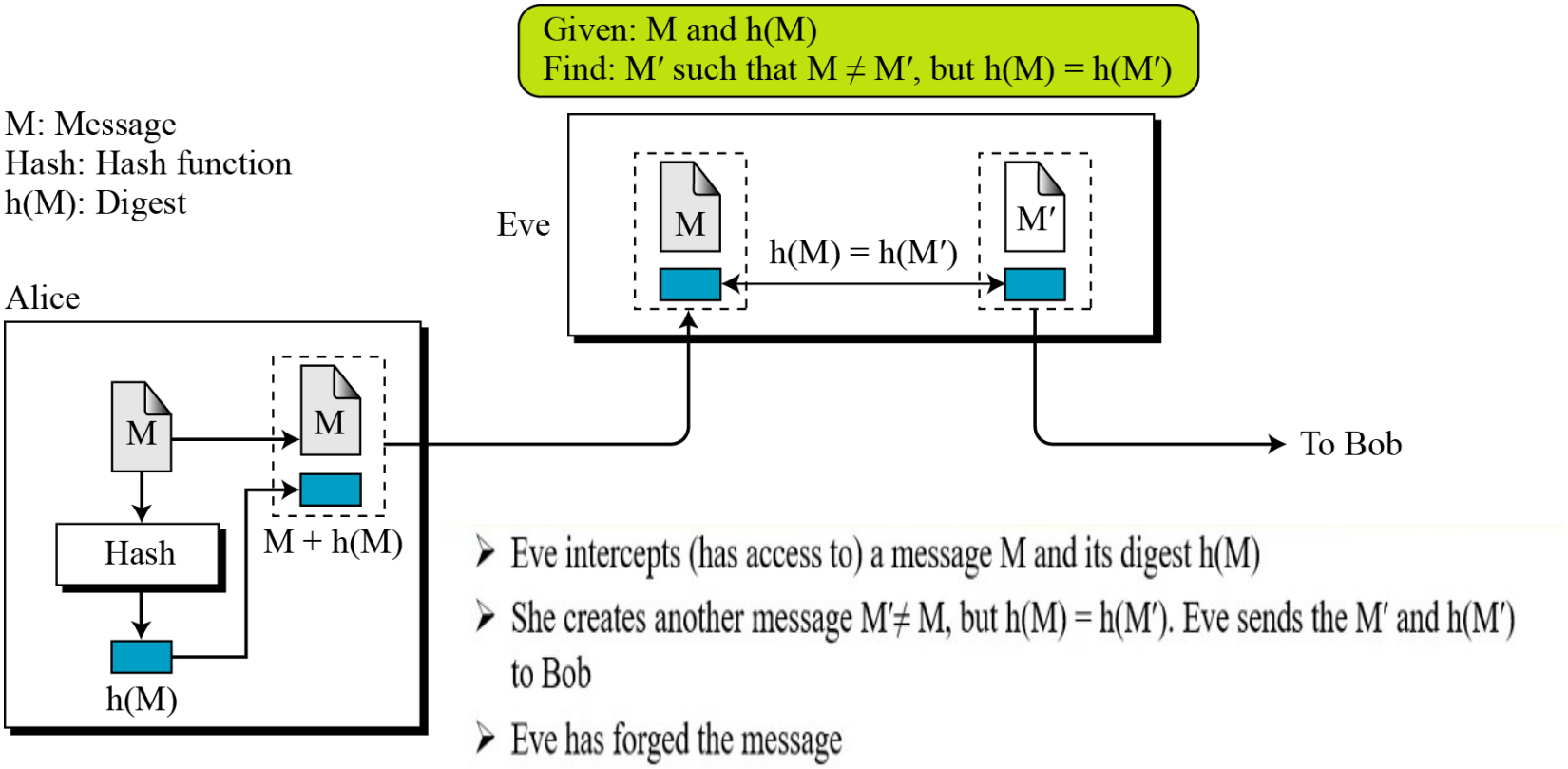
Preimage Resistance

Preimage Attack

Given: $y = h(M)$

Find: M' **such that** $y = h(M')$

Second Preimage Resistance



- 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 Attack

Given: M and h(M) **Find: M′ ≠ M such that h(M) = h(M′)**

Second Preimage Resistance

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.

Collision Resistance

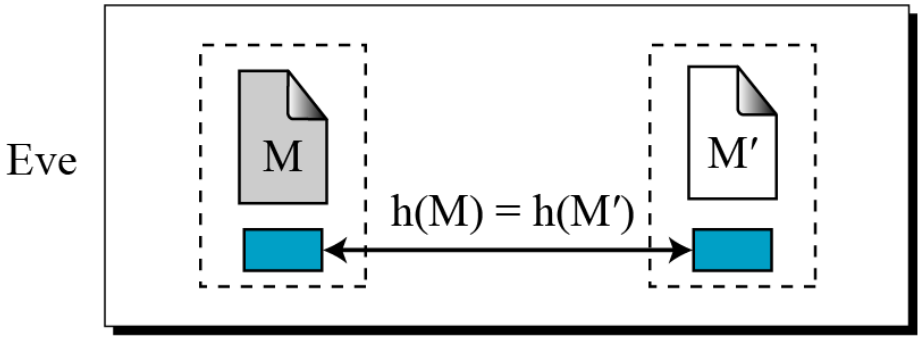
Ensures that Eve cannot find two messages that hash to the same digest.

Here the adversary can create two messages (out of scratch) and hashed to the same digest.

Collision Resistance

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

Find: M and M′ such that M ≠ M′, but h(M) = h(M′)



Collision Attack

Given: none

Find: M′ ≠ M such that h(M) = h(M′)

11.2 RANDOM ORACLE MODEL

is an ideal mathematical model for a hash function.

Introduced in 1993 by Bellare and Rogaway,

A function based on this model behaves as follows:

1. When a new message of any length is given, the oracle creates and gives a fixed length message digest that is a random string of 0s and 1s.

2. When a message is given for which a digest exists, the oracle simply gives the digest in the record.

3. The digest for a new message needs to be chosen independently from all previous digests. oracle cannot use a formula or an algorithm to calculate the digest.

11.2 RANDOM ORACLE MODEL

Table 11.1 *Oracle table after issuing the first three digests*

Message	*Message Digest*
4523AB1352CDEF45126	13AB
723BAE38F2AB3457AC	02CA
AB45CD1048765412AAAB6662BE	A38B

Example : Assume an oracle with a table and a fair coin.

- left column shows the messages whose digests have been issued by the oracle.
- second column lists the digests created for those messages.

Assumption: the digest is always 16 bits regardless of the size of the message.

11.2 RANDOM ORACLE MODEL

Now assume that two events occur:

a. The message AB1234CD8765BDAD is given for digest calculation.
   - ✓ Oracle checks its table. This message is not in the table, so the oracle flips its coin 16 times.
   - ✓ Assume that result is HHTHHHTTHTHHTTTH.
   - ✓ The oracle interprets H as a 1-bit and T as a 0-bit and gives 1101110010110001 in binary, or in hexadecimal, as the message digest for this message and adds message and digest in the table.

Table 11.1 *Oracle table after issuing the first three digests*

Message	*Message Digest*
4523AB1352CDEF45126	13AB
723BAE38F2AB3457AC	02CA
AB45CD1048765412AAAB6662BE	A38B

Table 11.2 *Oracle table after issuing the fourth digest*

Message	*Message Digest*
4523AB1352CDEF45126	13AB
723BAE38F2AB3457AC	02CA
AB1234CD8765BDAD	DCB1
AB45CD1048765412AAAB6662BE	A38B

11.2 RANDOM ORACLE MODEL

b. The message 4523AB1352CDEF45126 is given for digest calculation.

- ✓ The oracle checks its table and finds that there is a digest for this message in the table (first row).
- ✓ The oracle simply gives the corresponding digest (13AB).

Table 11.2 *Oracle table after issuing the fourth digest*

Message	*Message Digest*
4523AB1352CDEF45126	13AB
723BAE38F2AB3457AC	02CA
AB1234CD8765BDAD	DCB1
AB45CD1048765412AAAB6662BE	A38B

11.2 RANDOM ORACLE MODEL

Oracle in Example 11.3 cannot use a formula or algorithm to create the digest for a message.

For example, imagine the oracle uses the formula h(M) = M mod n.

Now suppose that the oracle has already given $h(M_1)$ and $h(M_2)$.

If a new message is presented as $M_3 = M_1 + M_2$, the oracle does not have to calculate the $h(M_3)$.
The new digest is just $[h(M_1) + h(M_2)] \bmod n$ since

$$h(M3) = (M1 + M2) \bmod n = M1 \bmod n + M2 \bmod n = [h(M1) + h(M2)] \bmod n$$

This violates the third requirement that each digest must be randomly chosen based on the message given to the oracle.

11.3 MESSAGE AUTHENTICATION

A message digest guarantees the integrity of a message.

It guarantees that the message has not been changed.

A message digest, however, 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.

11.3 MESSAGE AUTHENTICATION

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.

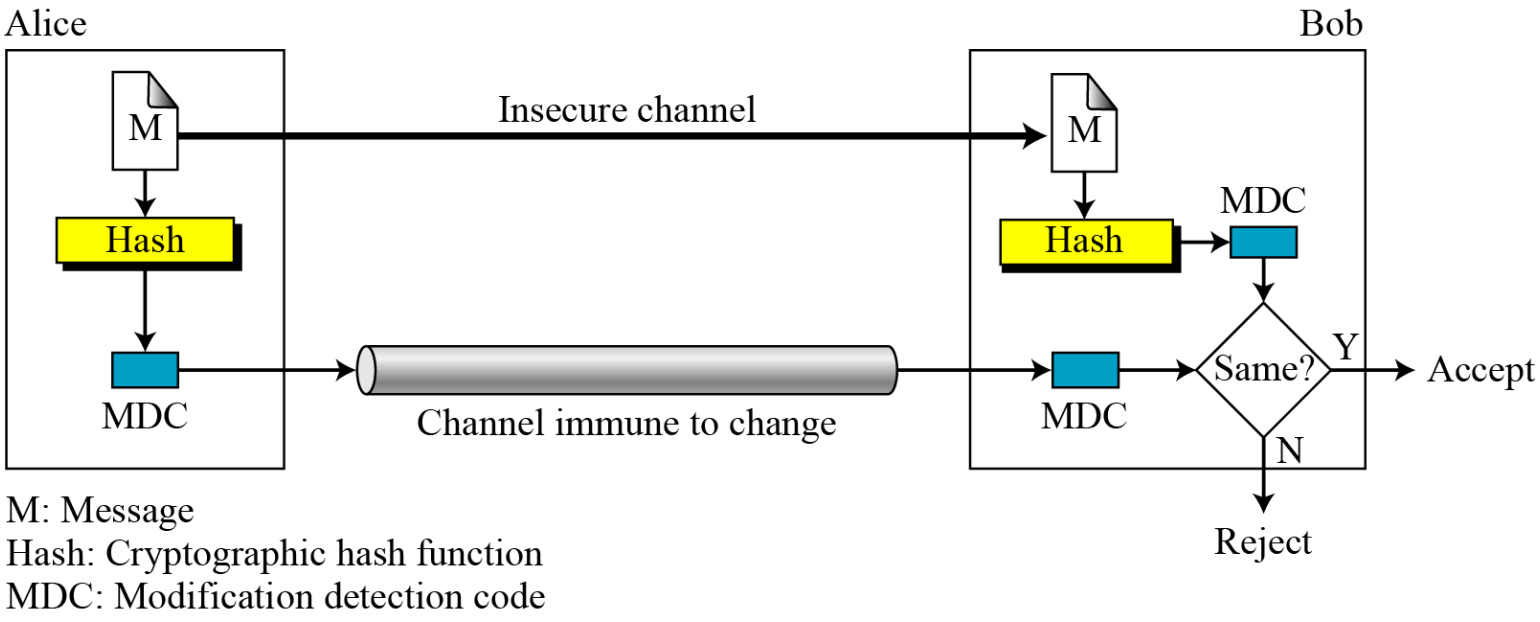
What we need for message authentication (data origin authentication) is a message authentication code (MAC).

Modification Detection Code (MDC)

A modification detection code (MDC) 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.

Modification Detection Code (MDC)



Modification Detection Code (MDC)

Figure 11.9 shows that 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.

Note that the term safe can mean a trusted party;

Modification Detection Code (MDC)

Alice writes her will and announces it publicly (insecure channel).

Alice makes an MDC from the message and deposits it with her attorney, which is kept until her death (a secure channel).

Although Eve may change the contents of the will, the attorney can create an MDC from the will and prove that Eve's version is a forgery.

If the cryptography hash function used to create the MDC has the three properties described at the beginning of this chapter, Eve will lose.

Message Authentication Code (MAC)

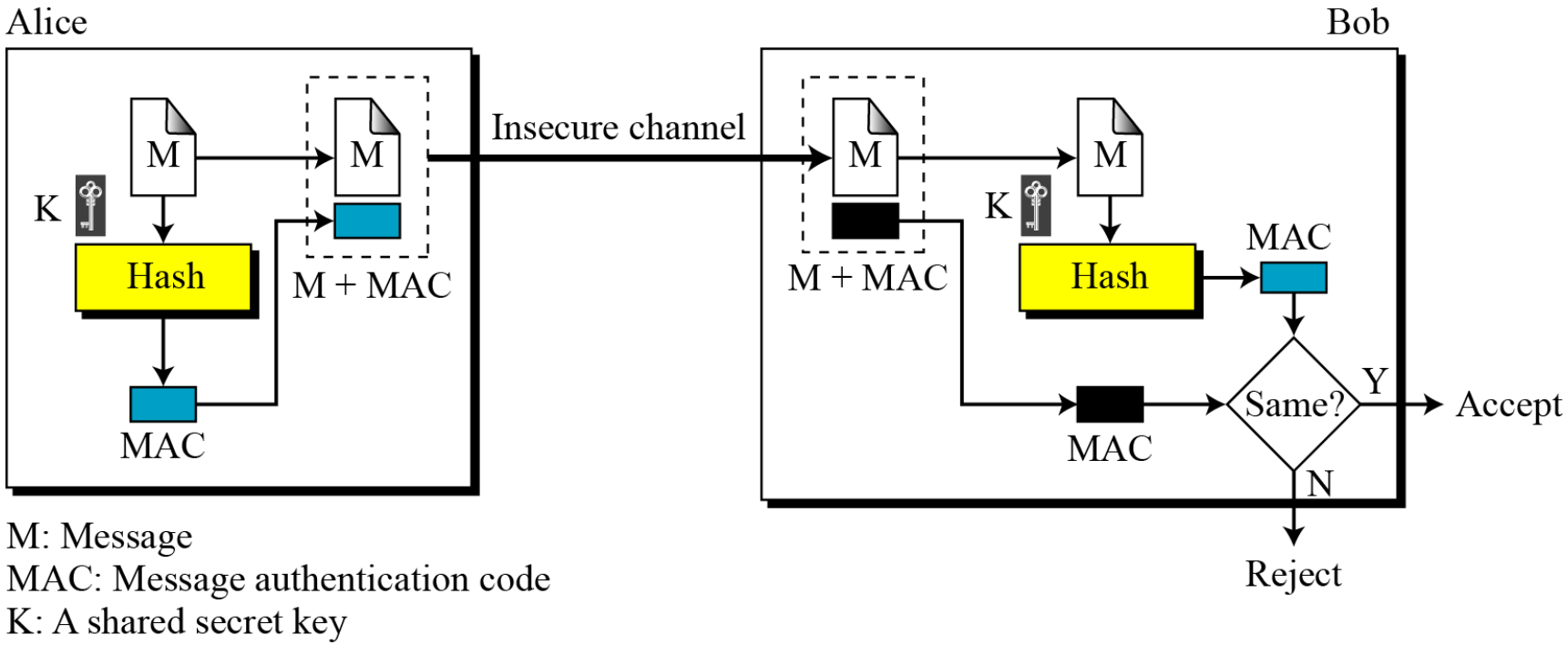
To ensure the integrity of the message and the data origin authentication

ie Alice is the originator of the message, not somebody else

we need to change a modification detection code (MDC) to a message authentication code (MAC).

The difference between a MDC and a MAC is that the second includes a secret between Alice and Bob.

Message Authentication Code (MAC)



M: Message
MAC: Message authentication code
K: A shared secret key

Message Authentication Code (MAC)

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 2 MACs match, message is authentic and has not been modified by an adversary.

Message Authentication Code (MAC)

There is no need to use two channels in this case.

Message Authentication Code (MAC)

The MAC described is referred to as a prefix MAC because the secret key is appended to the beginning of the message.

We can have a postfix MAC, in which the key is appended to the end of the message.

We can combine the prefix and postfix MAC, with the same key or two different keys.

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

Nested MAC

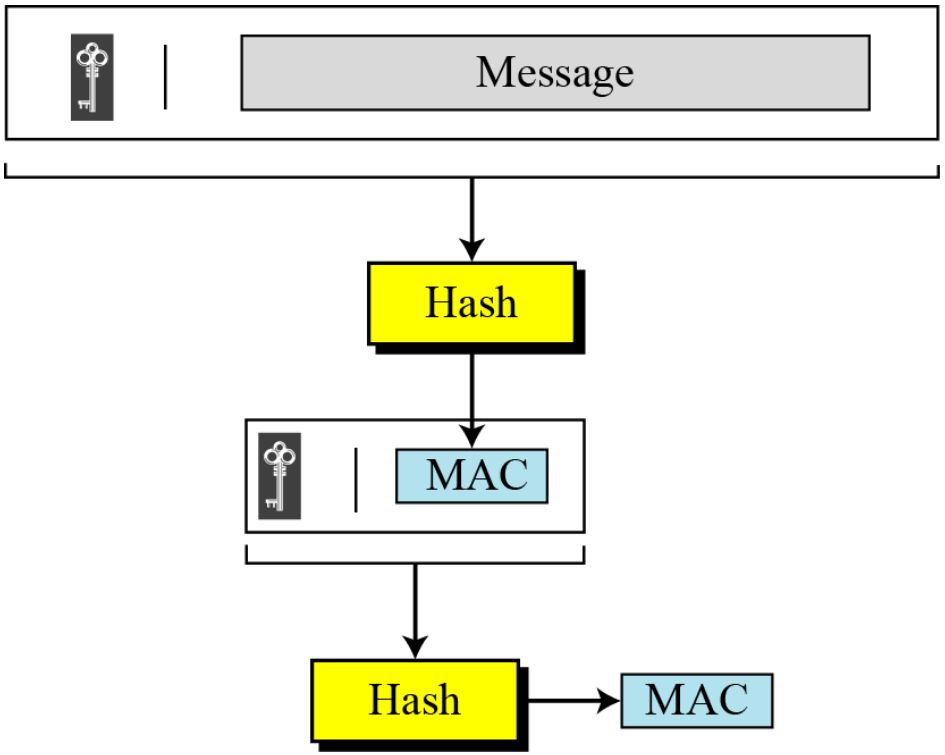


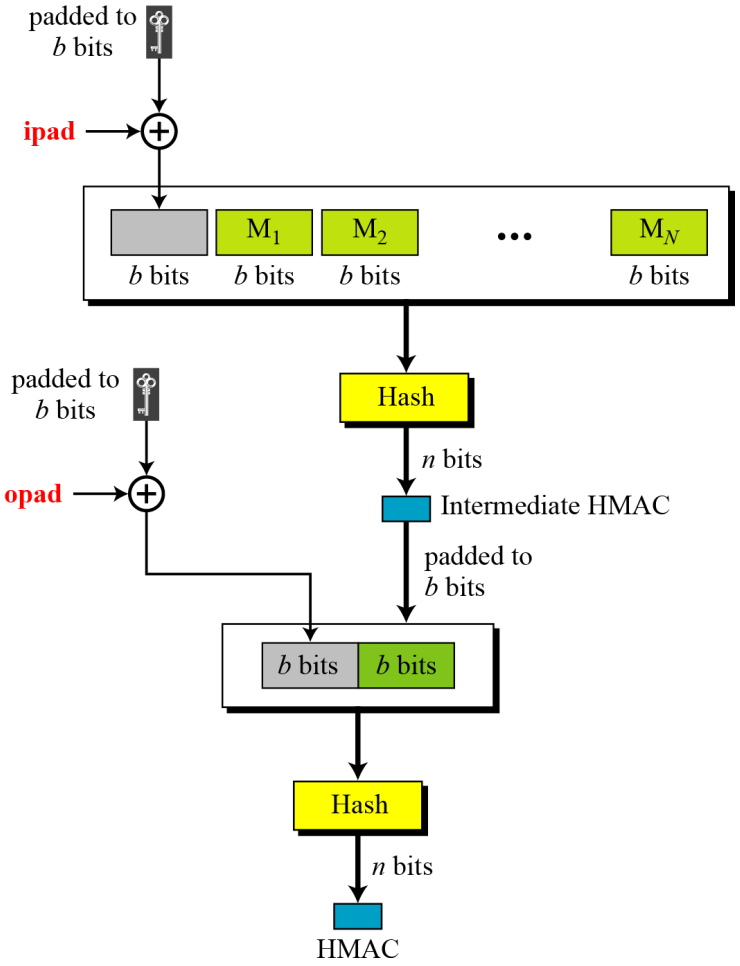
Figure 11.11 *Nested MAC*

To improve the security of a MAC, nested MACs were designed in which 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.

HMAC

NIST has issued a standard for a nested MAC that is often referred to as HMAC.



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. It is recommended that 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. Value of ipad is the b/8 repetition of the sequence 00110110 (36 in hexadecimal).

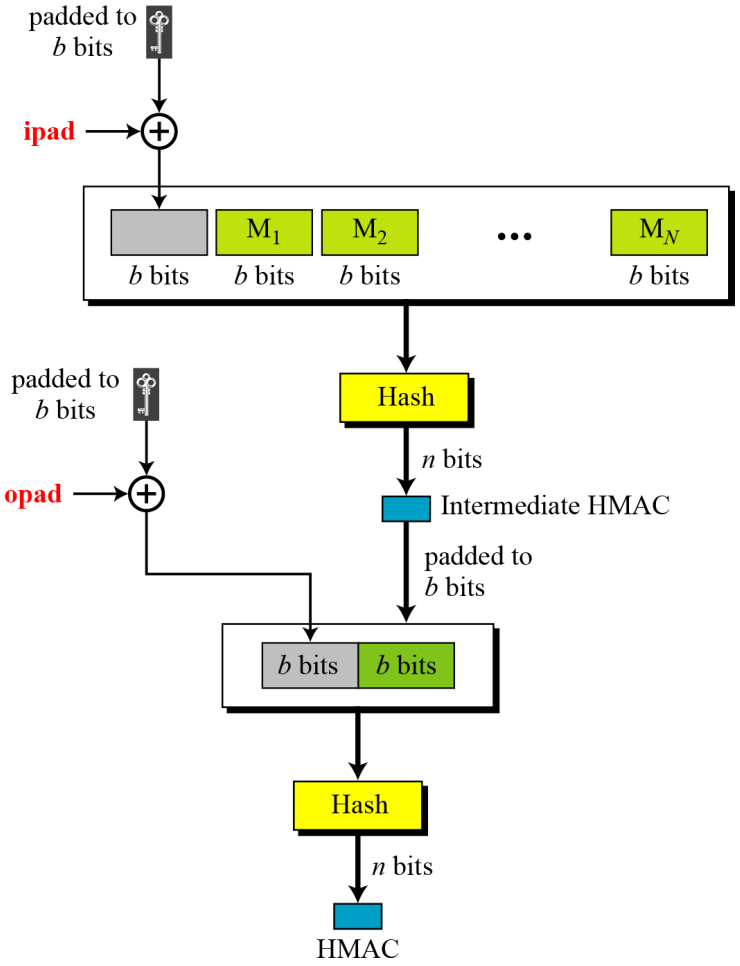
4. The resulting block is prepended to the N-block message. The result is N + 1 blocks.

5. Result of step 4 is hashed to create an n-bit digest (intermediate HMAC)

Hash based message authentication code (HMAC)

HMAC

NIST has issued a standard for a nested MAC that is often referred to as HMAC.



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

Hash Function

12.1 INTRODUCTION

Cryptographic hash function takes a message of arbitrary length and creates a message digest of fixed length.

The two most promising cryptographic hash algorithms- SHA-512 and Whirlpool.

Before discuss some general ideas that may be applied to any cryptographic hash function.

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.

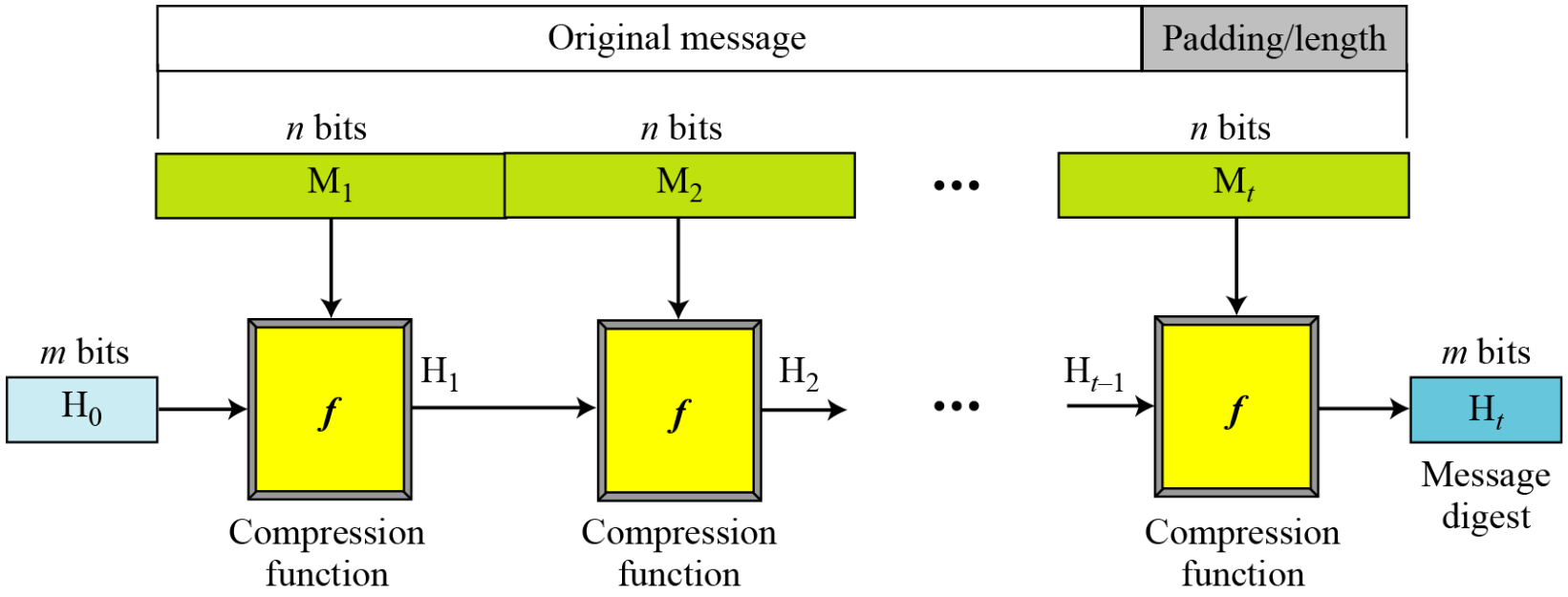
The fixed-size input function is referred to as a compression function.

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.



Merkle-Damgard Scheme (contd)

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. Call each block M_1, M_2,…, M_t.

   digest created at t iterations H_1, H_2,…, H_t.

3. Before starting the iteration, the digest H_0 is set to a fixed value, normally called IV (initial value or initial vector).

4. The compression function at each iteration operates on H_{i-1} and Mi to create a new H_i.

   $H_i = f(H_{i-1}, M_i)$, where f is the compression function.

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

If the compression function in the Merkle-Damgard scheme is collision resistant, the hash function is also collision resistant.

Two Groups of Compression Functions

Merkle-Damgard scheme is the basis for many cryptographic hash functions today.

Only thing we need to do is design a compression function that is collision resistant and insert it in the Merkle-Damgard scheme.

Two different approaches in designing a hash function.

- First approach : compression function is made from scratch: it is particularly designed for this purpose.
- Second approach : symmetric-key block cipher serves as a compression function.

Hash Functions Made from Scratch

A set of cryptographic hash functions uses compression functions that are made from scratch.

These compression functions are specifically designed for the purposes they serve.

Message Digest (MD)

- ✓ Several hash algorithms were designed by Ron Rivest.
- ✓ These are referred to as MD2, MD4, and MD5.
- ✓ 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.
- ✓ 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).
- ✓ It is sometimes referred to as Secure Hash Standard (SHS).
- ✓ The standard is mostly based on MD5.
- ✓ The standard was revised in 1995, which includes SHA-1.
- ✓ It was revised later, which defines 4 new versions: SHA-224, SHA-256, SHA-384, and SHA-512.

Table 12.1 *Characteristics of Secure Hash Algorithms (SHAs)*

Characteristics	*SHA-1*	*SHA-224*	*SHA-256*	*SHA-384*	*SHA-512*
Maximum Message size	$2^{64} - 1$	$2^{64} - 1$	$2^{64} - 1$	$2^{128} - 1$	$2^{128} - 1$
Block size	512	512	512	1024	1024
Message digest size	160	224	256	384	512
Number of rounds	80	64	64	80	80
Word size	32	32	32	64	64

All of these versions have the same structure

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.

There are several secure symmetric-key block ciphers, such as triple DES or AES, that can be used to make a one-way function.

The block cipher in this case only performs encryption.

Several schemes have been proposed.

one of the most promising, Whirlpool.

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;
Previously created digest is used as the plaintext. The ciphertext is the new message digest.

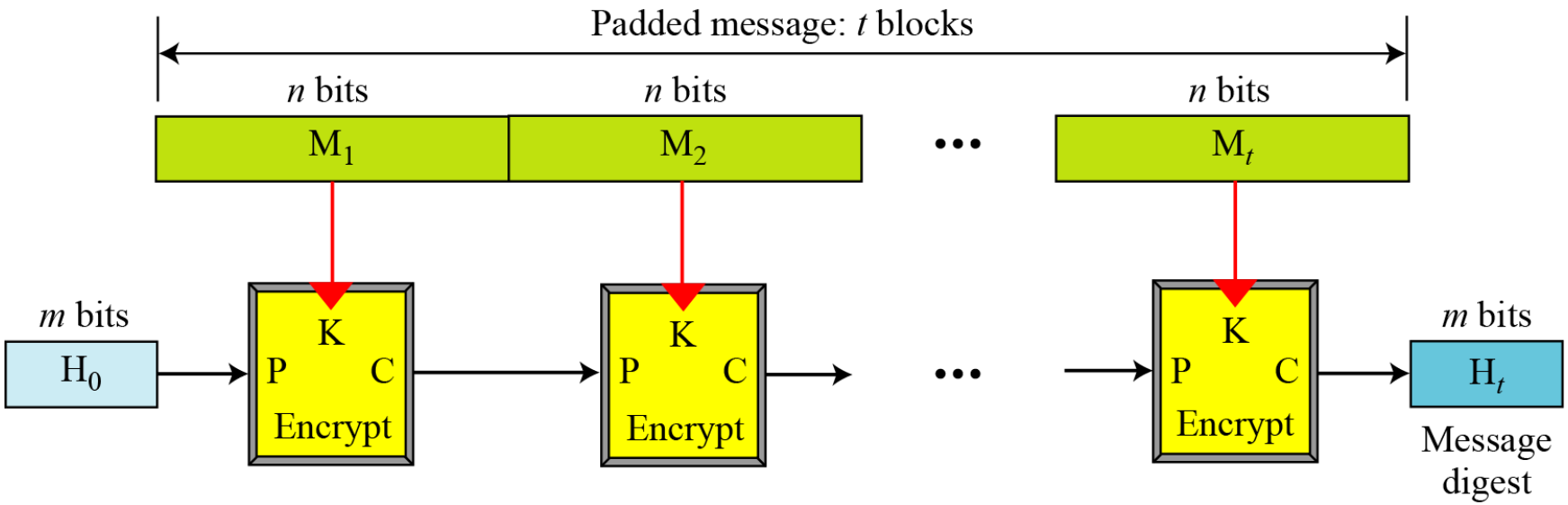
The digest is the size of data block cipher in the underlying cryptosystem.

For example, if DES is used as the block cipher, the size of the digest is only 64 bits.

it is subject to a meet-in-the-middle attack, because the adversary can use the decryption algorithm of the cryptosystem.

Rabin Scheme

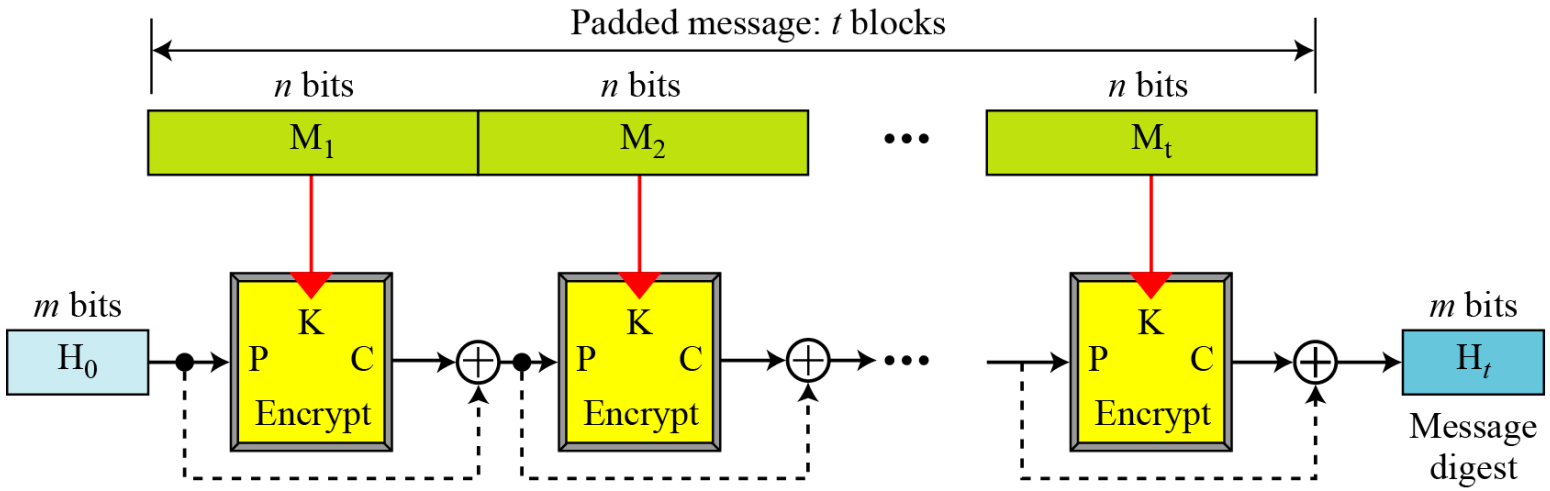
Figure 12.2 *Rabin scheme*



Davies-Meyer Scheme

The Davies-Meyer scheme is basically the same as the Rabin scheme except that it uses forward feed to protect against meet-in-the-middle attack.

Figure 12.3 *Davies-Meyer scheme*

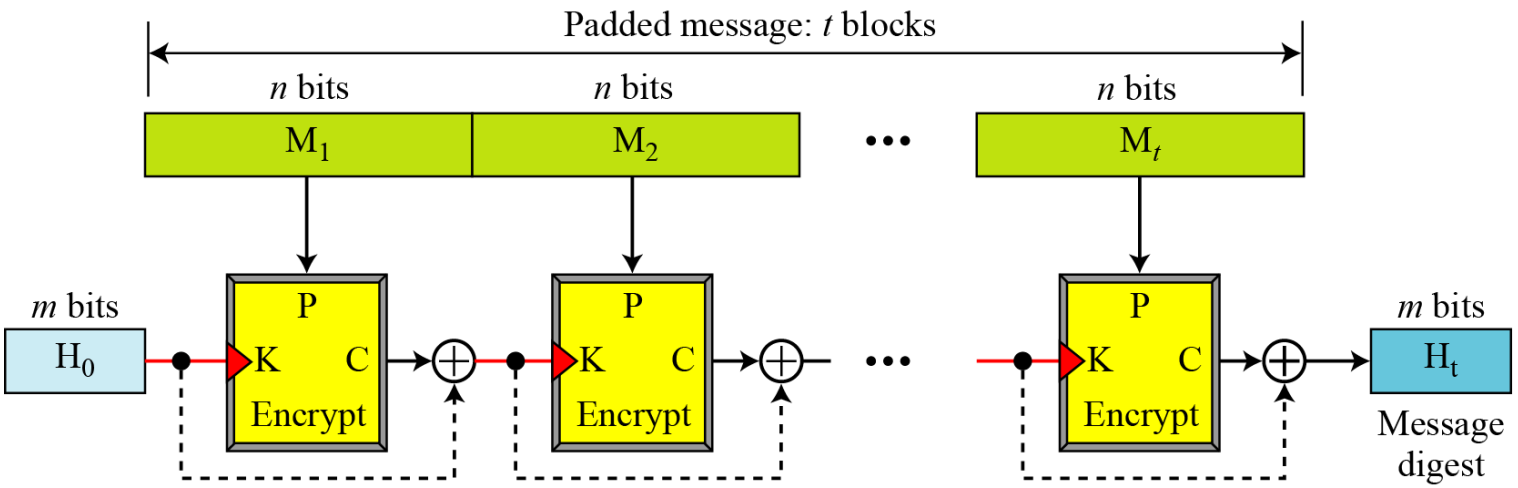


Matyas-Meyer-Oseas Scheme

The scheme can be used if the data block and the cipher key are the same size.

For example, AES is a good candidate for this purpose.

Figure 12.4 *Matyas-Meyer-Oseas scheme*



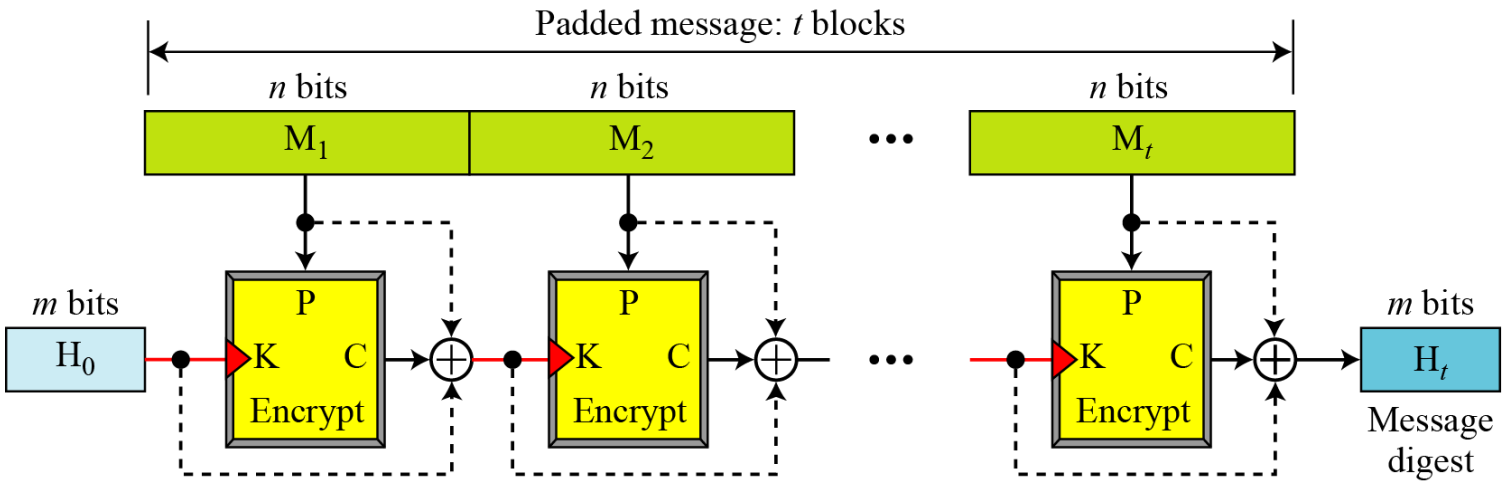
Miyaguchi-Preneel Scheme

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.

Figure 12.5 *Miyaguchi-Preneel scheme*



12.2 SHA-512

SHA-512 is the version of SHA with a 512-bit message digest.

This version, like others SHA family, is based on the Merkle-Damgard scheme.

it is the latest version, it has a more complex structure than the others, and its message digest is the longest.

12.2 SHA-512

For characteristics of SHA-512 see Table 12.1.

Table 12.1 *Characteristics of Secure Hash Algorithms (SHAs)*

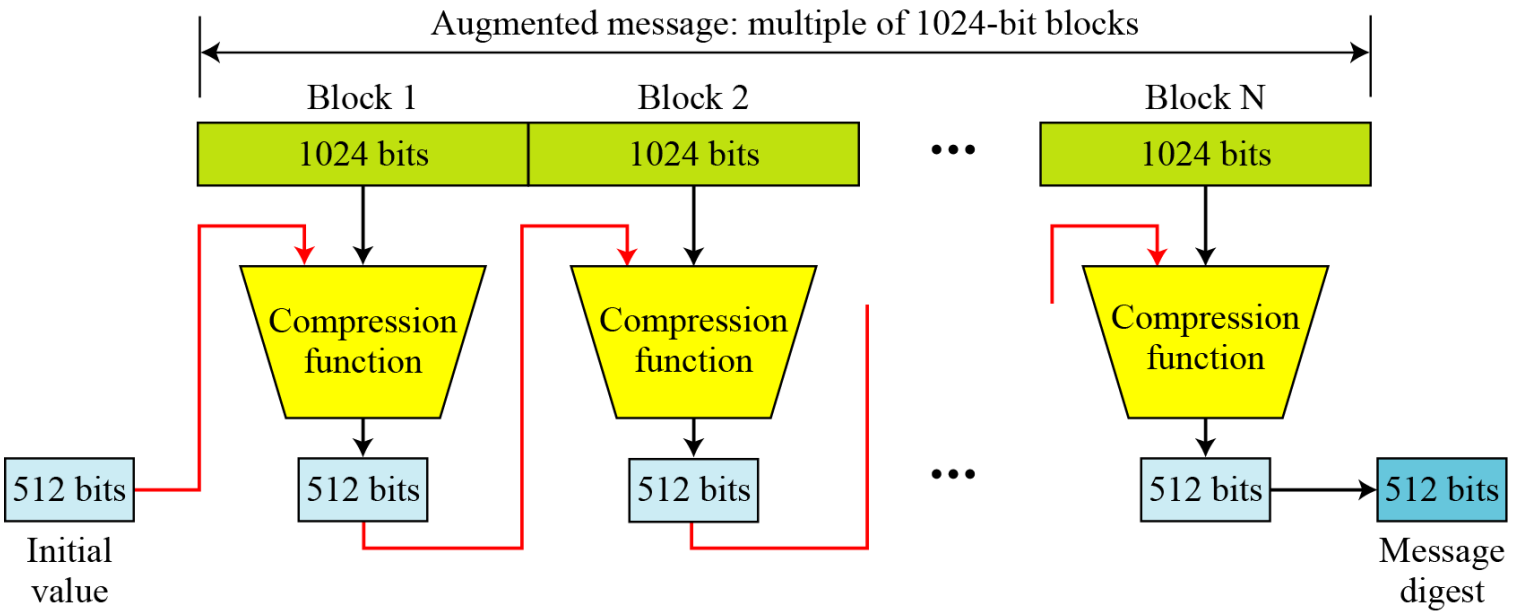
Characteristics	*SHA-1*	*SHA-224*	*SHA-256*	*SHA-384*	*SHA-512*
Maximum Message size	$2^{64} - 1$	$2^{64} - 1$	$2^{64} - 1$	$2^{128} - 1$	$2^{128} - 1$
Block size	512	512	512	1024	1024
Message digest size	160	224	256	384	512
Number of rounds	80	64	64	80	80
Word size	32	32	32	64	64

12.2 SHA-512

SHA-512 creates a digest of 512 bits from a multiple-block message.

Each block is 1024 bits in length.

Figure 12.6 *Message digest creation SHA-512*



12.2 SHA-512

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 N^{th} digest.

When the last block is processed, the resulting digest is the message digest for the entire message.

12.2 SHA-512

Message Preparation

SHA-512 insists that the length of the original message be less than 2^{128} bits.

This means that if the length of a message is equal to or greater than 2^{128}, it will not be processed by SHA-512.

This is not usually a problem because 2^{128} bits is probably larger than the total storage capacity of any system.

12.2 SHA-512

Message Preparation

SHA-512 creates a 512-bit message digest out of a message less than 2^{128}.

12.2 SHA-512

Length Field and Padding

Before the message digest can be created, 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.

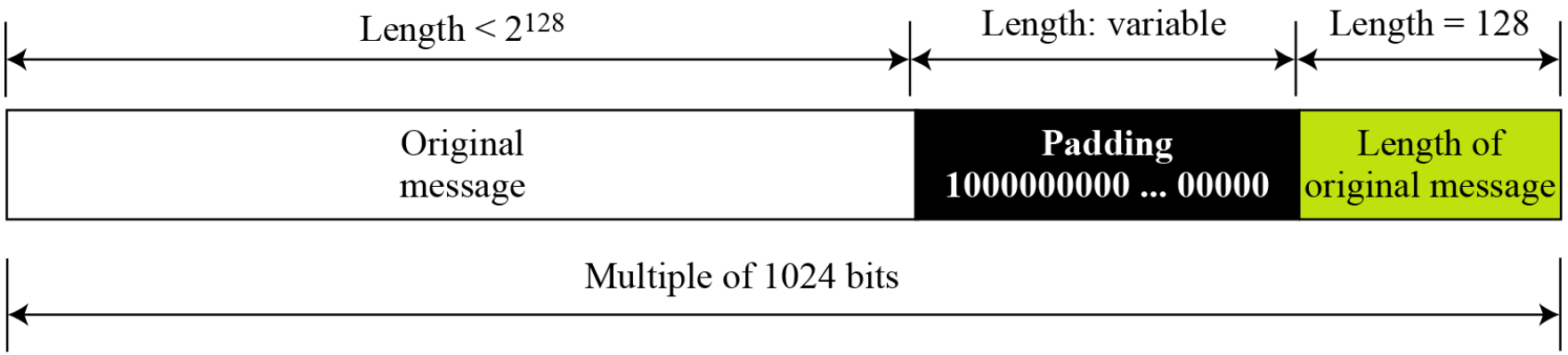
An unsigned integer field of 128 bits can define a number between 0 and $2^{128} - 1$, which is the maximum length of the message allowed in SHA-512.

The length field defines the length of the original message before adding the length field or the padding (Figure 12.7).

12.2 SHA-512

Length Field and Padding

Figure 12.7 ***Padding and length field in SHA-512***



12.2 SHA-512

Length Field and Padding

Before the addition of the length field, we need to pad the original message to make the length a multiple of 1024.

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.

12.2 SHA-512

Length Field and Padding

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

Solution

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

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

12.2 SHA-512

Length Field and Padding

Example 12.4
Do we need padding if the length of the original message is already a multiple of 1024 bits?

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

12.2 SHA-512

Example 12.5

What is the minimum and maximum number of padding bits that can be added to a message?

Solution

a. The minimum length of padding is 0 and it happens when $(-M - 128) \bmod 1024$ is 0. This means that $|M| = -128 \bmod 1024 = 896 \bmod 1024$ bits. In other words, the last block in the original message is 896 bits. We add a 128-bit length field to make the block complete.

b. The maximum length of padding is 1023 and it happens when $(-|M| - 128) = 1023 \bmod 1024$. This means that the length of the original message is $|M| = (-128 - 1023) \bmod 1024$ or the length is $|M| = 897 \bmod 1024$. In this case, we cannot just add the length field because the length of the last block exceeds one bit more than 1024. So we need to add 897 bits to complete this block and create a second block of 896 bits. Now the length can be added to make this block complete.

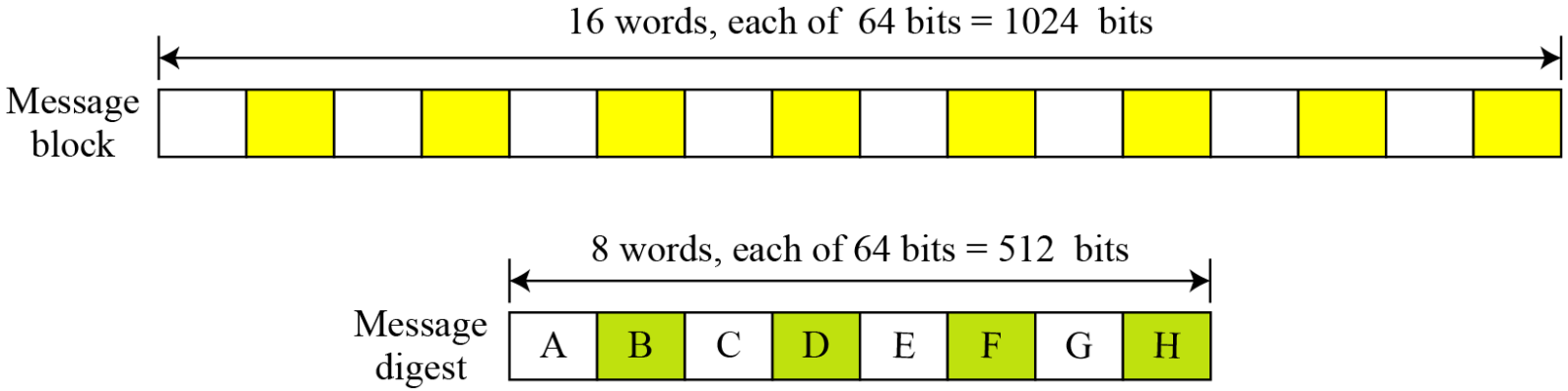
12.2 SHA-512

Words

SHA-512 operates on words; A word is defined as 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.

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.



12.2 SHA-512

Words

SHA-252 is word-oriented. Each block is 16 words; the digest is only 8 words.

16*64=1024
8*64=512

12.2 SHA-512

Word Expansion

- Before processing, each message block must be expanded.
- A block is made of 1024 bits, or sixteen 64-bit words.
- We need 80 words in the processing phase.
- So the 16-word block needs to be expanded to 80 words, from W_0 to W_{79}.

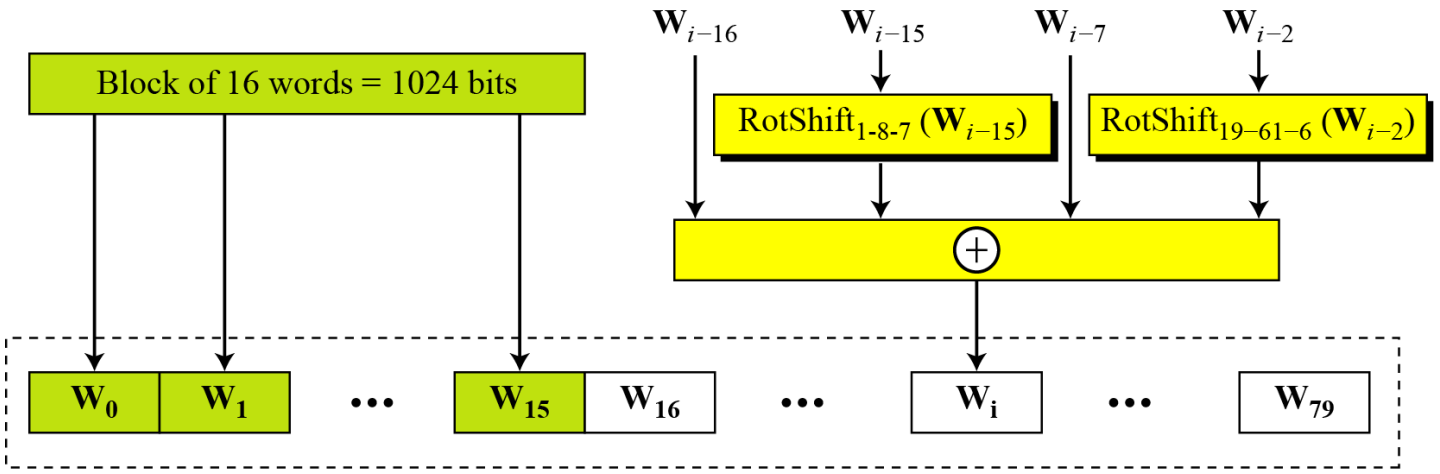
12.2 SHA-512

Word Expansion

The 1024-bit block becomes the first 16 words;

The rest of the words come from already-made words according to the operation shown in the figure.

Figure 12.9 *Word expansion in SHA-512*



$RotShift_{l\text{-}m\text{-}n}(x)$: $RotR_l(x) \oplus RotR_m(x) \oplus ShL_n(x)$

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

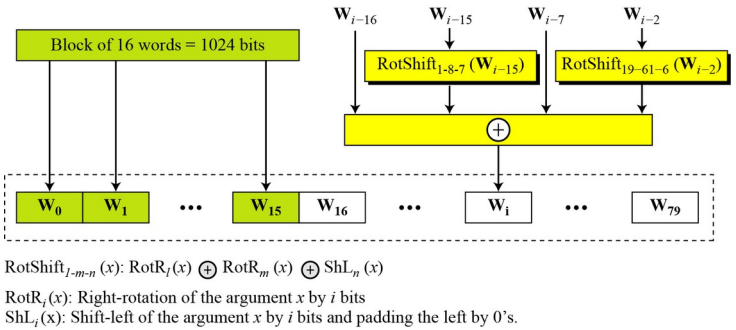
$ShL_i(x)$: Shift-left of the argument x by i bits and padding the left by 0's.

12.2 SHA-512

Example 12.6
Show how W_{60} is made.

Solution

$$W_{60} = W_{44} \oplus \text{RotShift}_{1\text{-}8\text{-}7}(W_{45}) \oplus W_{53} \oplus \text{RotShift}_{19\text{-}61\text{-}6}(W_{58})$$



$\text{RotShift}_{l\text{-}m\text{-}n}(x)$: $\text{RotR}_l(x) \oplus \text{RotR}_m(x) \oplus \text{ShL}_n(x)$

$\text{RotR}_i(x)$: Right-rotation of the argument x by i bits
$\text{ShL}_i(x)$: Shift-left of the argument x by i bits and padding the left by 0's.

12.2 SHA-512

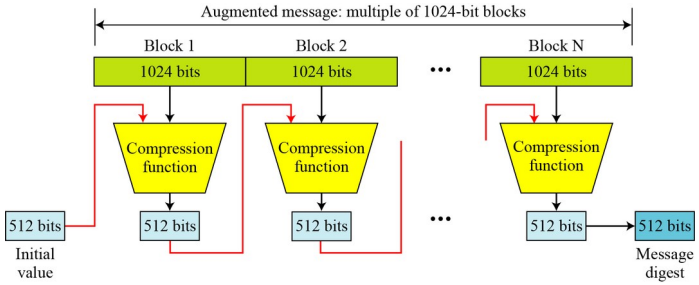
Message Digest Initialization

The algorithm uses 8 constants for message digest initialization.

Call these constants A_0 to H_0 to match with the word naming used for the digest.

Table 12.2 *Values of constants in message digest initialization of SHA-512*

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



12.2 SHA-512

Values are calculated from the first 8 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.

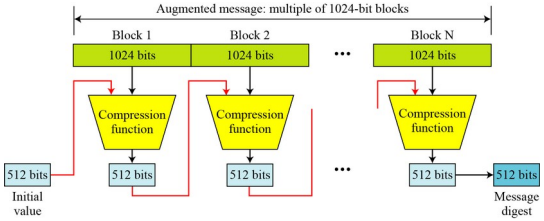
For example, the eighth prime is 19, with the square root $(19)^{1/2} = 4.35889894354$.

Converting the number to binary with only 64 bits in the fraction part, we get

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

SHA-512 keeps the fraction part, $(5BE0CD19137E2179)_{16}$, as an unsigned integer

12.2 SHA-512



Compression Function

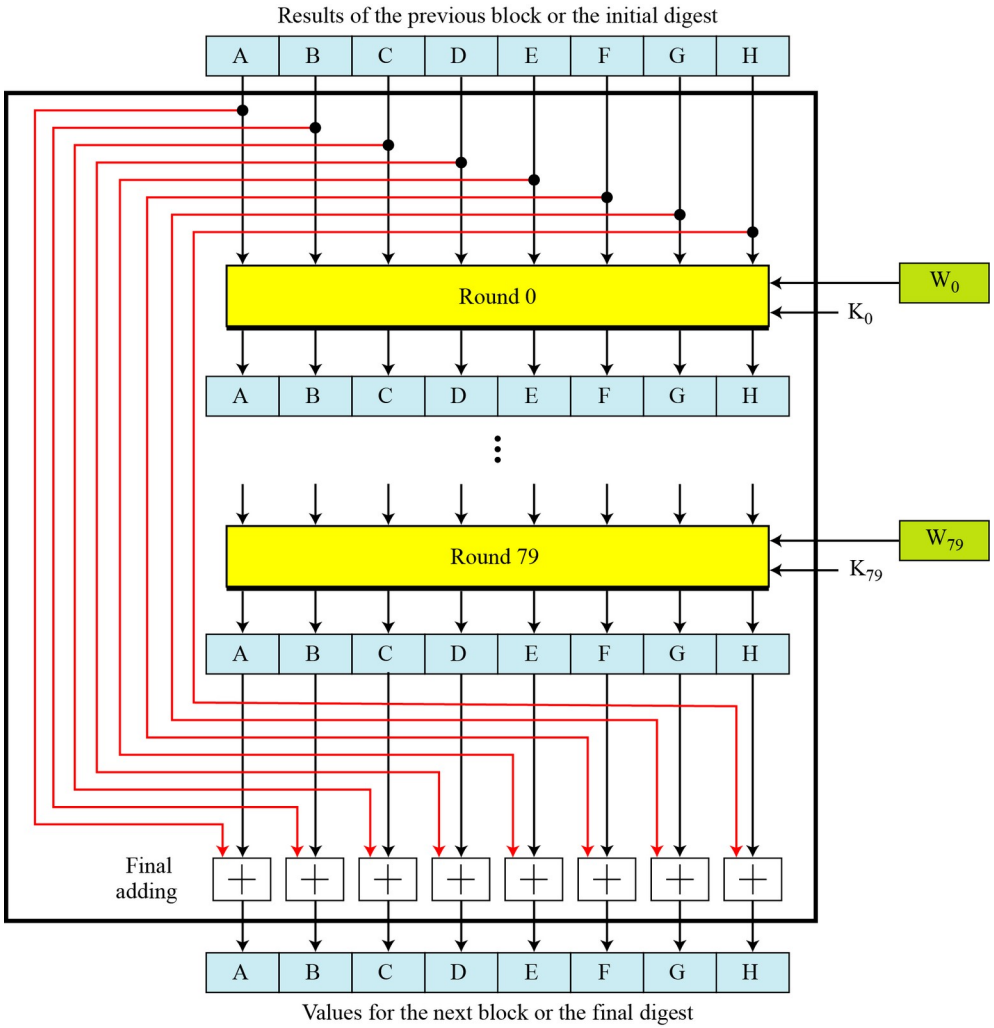
SHA-512 creates a 512-bit (eight 64-bit words) message digest from a multiple-block message where each block is 1024 bits.

The processing of each block of data in SHA-512 **involves 80 rounds.**

Figure 12.10 shows the general outline for the compression function.

12.2 SHA-512

Compression Function



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.

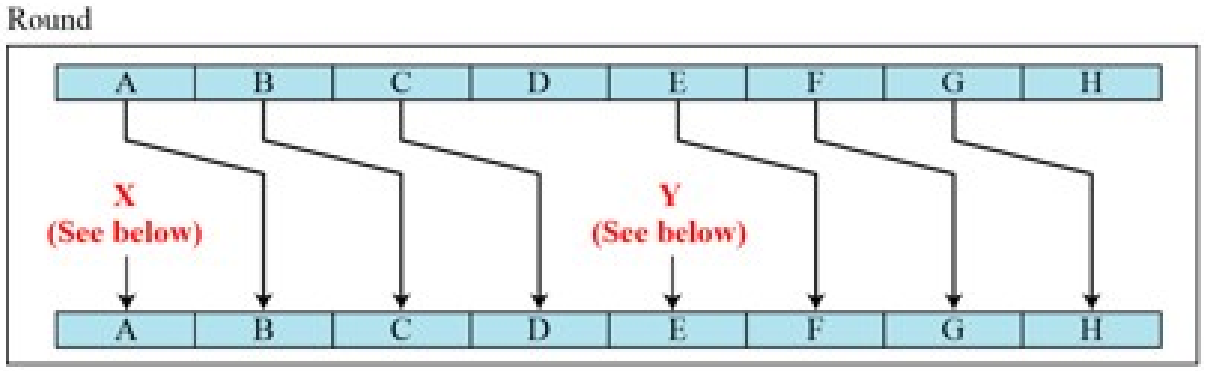
We call this last operation the final adding, as shown in the figure.

Structure of Each Round

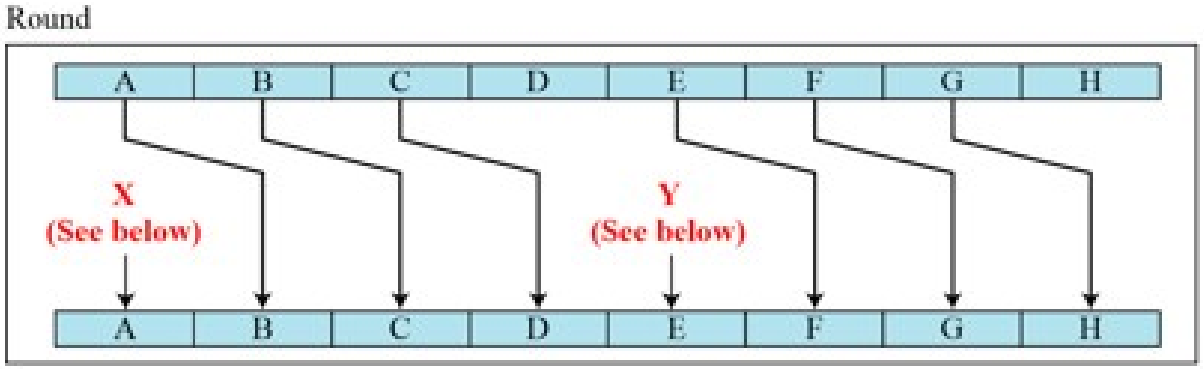
In each round, eight new values for the 64-bit buffers are created from the values of the buffers in the previous round.

As Figure 12.11 shows, six buffers are the exact copies of one of the buffers in the previous round as shown below:

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



Structure of Each Round

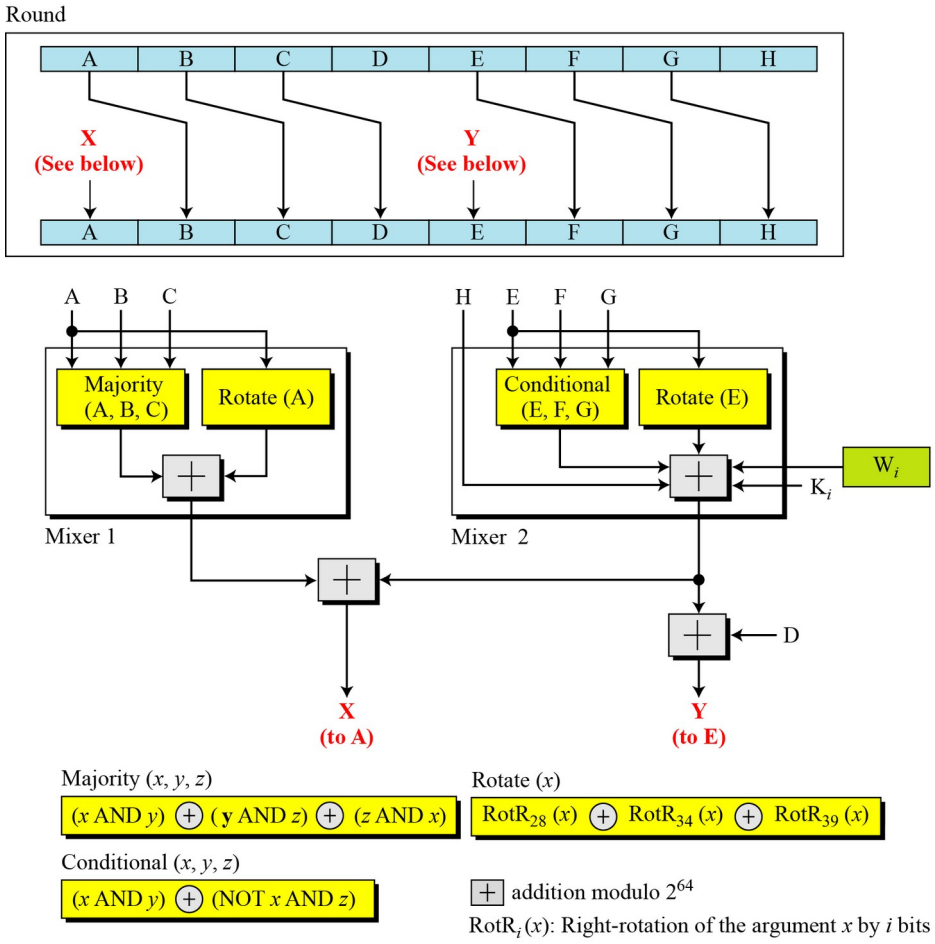


Two of the new buffers, A and E, receive their inputs from some complex functions that involve some of the previous buffers, the corresponding word for this round (Wi), and the corresponding constant for this round (Ki).

Figure 12.11 shows the structure of each round.

Structure of Each Round

Figure 12.11 shows the structure of each round.

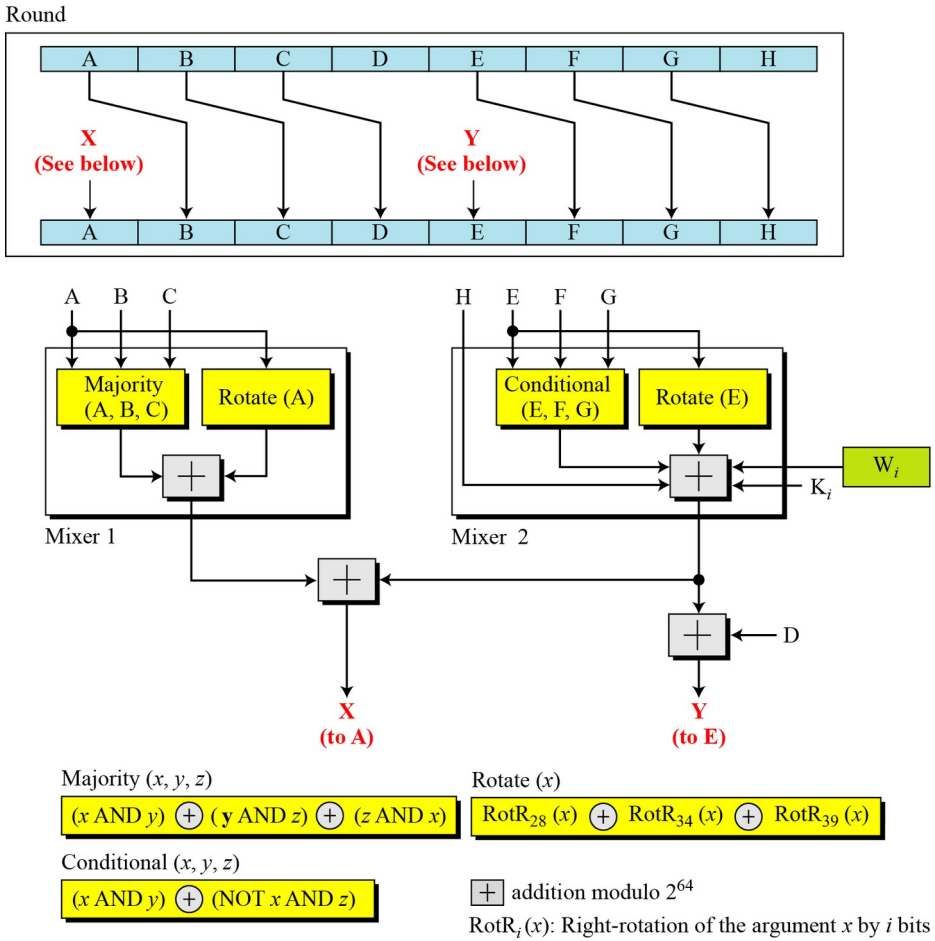


There are 2 mixers, 3 functions, and several operators.

Each mixer combines two functions.

Structure of Each Round

Figure 12.11 shows the structure of each round.

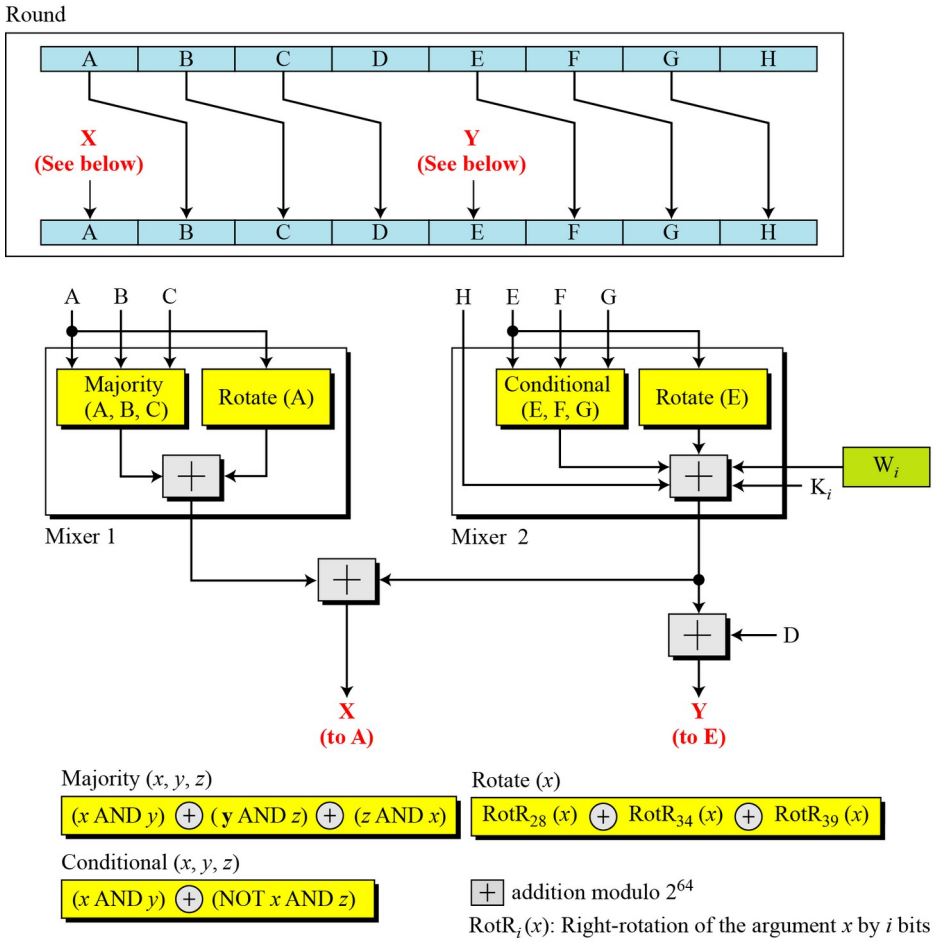


1. The Majority function is a bitwise function.
It takes three corresponding bits in three buffers (A, B, and C) and calculates

$$(A_j \text{ AND } B_j) \oplus (B_j \text{ AND } C_j) \oplus (C_j \text{ AND } A_j)$$

Resulting bit is the majority of 3 bits.
If 2 or 3 bits are 1's, the resulting bit is 1; otherwise it is 0.

Structure of Each Round

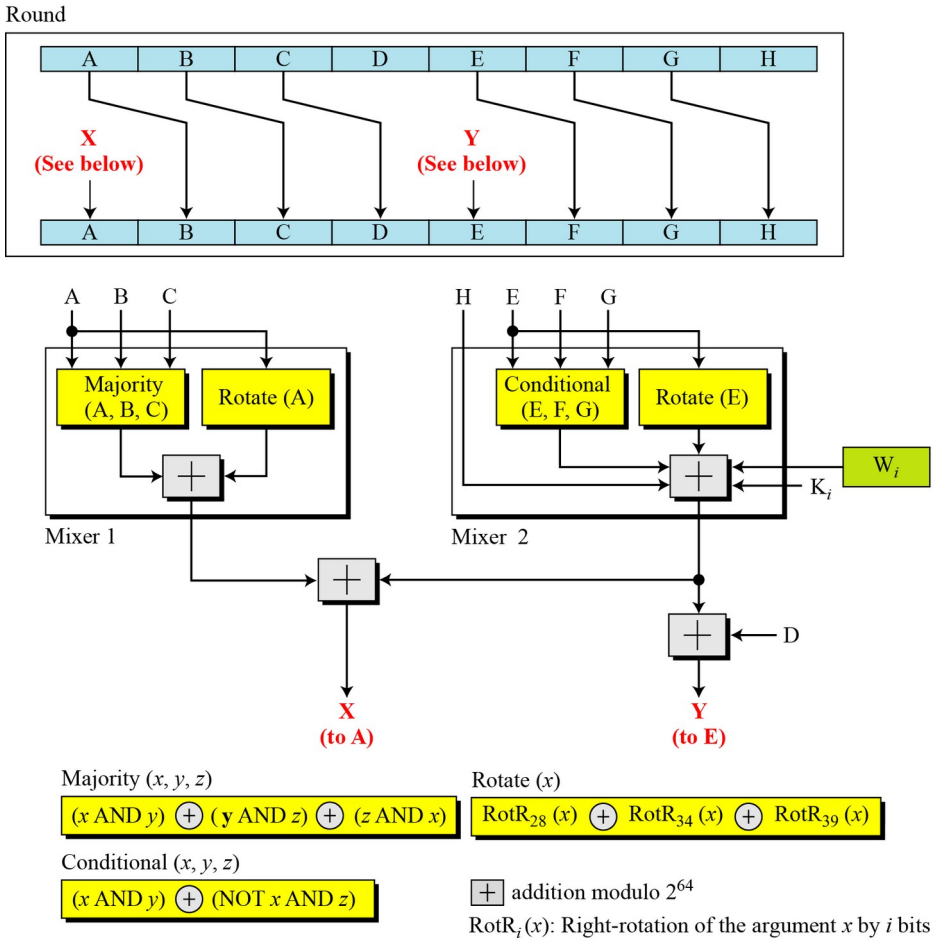


2. The Conditional function is also a bitwise function.
It takes three corresponding bits in three buffers (E, F, and G) and calculates

$$(E_j \text{ AND } F_j) \oplus (\text{NOT } E_j \text{ AND } G_j)$$

Resulting bit is the logic "If E_j then F_j; else G_j".

Structure of Each Round



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

Rotate (A): $\mathbf{RotR_{28}(A) \oplus RotR_{34}(A) \oplus RotR_{29}(A)}$

Rotate (E): $\mathbf{RotR_{28}(E) \oplus RotR_{34}(E) \oplus RotR_{29}(E)}$

4. The right-rotation function, $RotR_i(x)$, is the same as the one we used in the wordexpansion process.

It 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.

Structure of Each Round

6. There are 80 constants, K_0 to K_{79}, each of 64 bits as shown in Table 12.3 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).

Table 12.3 *Eighty constants used for eighty rounds in SHA-512*

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

Structure of Each Round

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.

For example, the 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 $(111.0110\ 1100\ 0100\ 0100\ \ldots\ 0111)_2 \rightarrow (7.6C44198C4A475817)_{16}$

SHA-512 keeps the fraction part, $(6C44198C4A475817)_{16}$, as an unsigned integer.

Structure of Each Round

Example 12.7
We apply the 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. We can also prove it 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.

Structure of Each Round

Example 12.8
We 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?

Solution
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 definition of the Condition function to prove the result:

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

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.

12.3 WHIRLPOOL

Is an iterated cryptographic hash function, based on the Miyaguchi-Preneel scheme.

Uses a symmetric-key block cipher in place of the compression function.

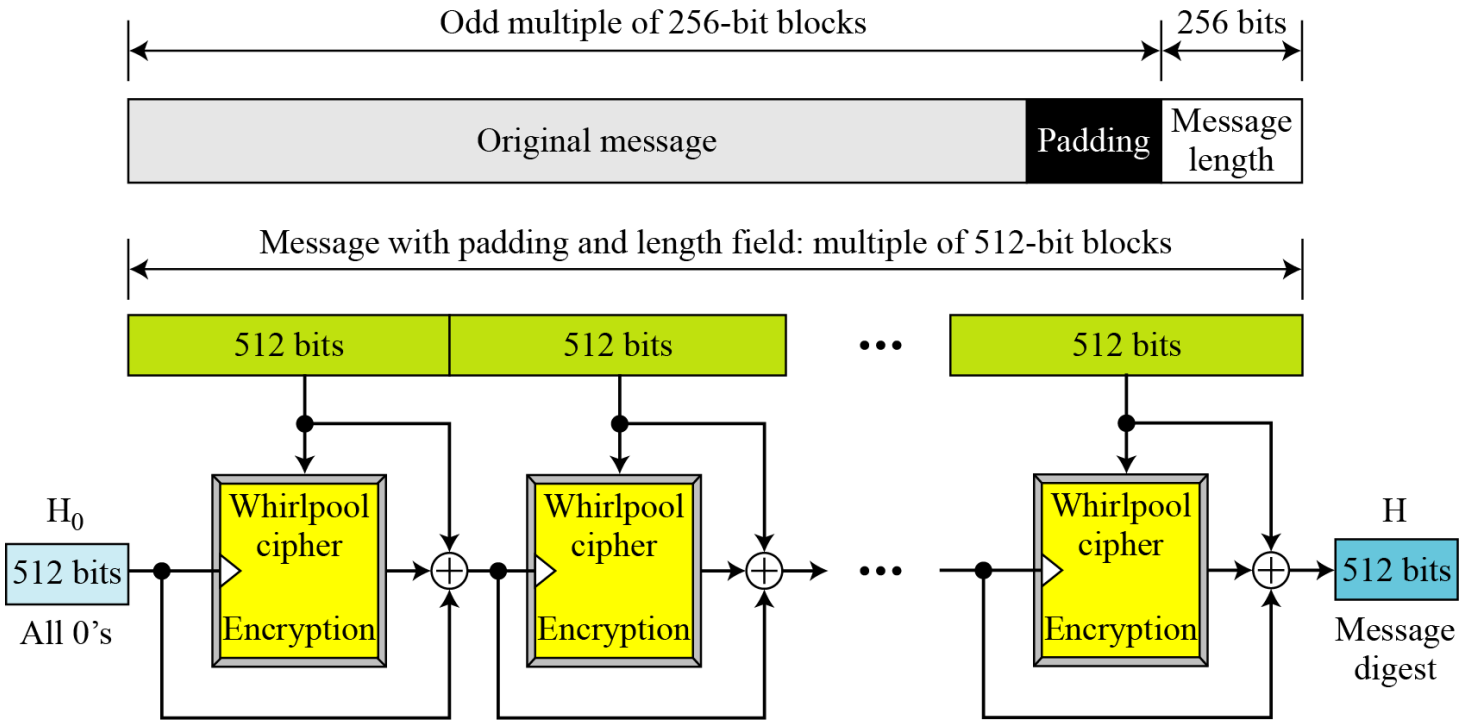
Block cipher is a modified AES cipher that has been tailored for this purpose.

Designed by Vincent Rijmen and Paulo S. L. M. Barreto.

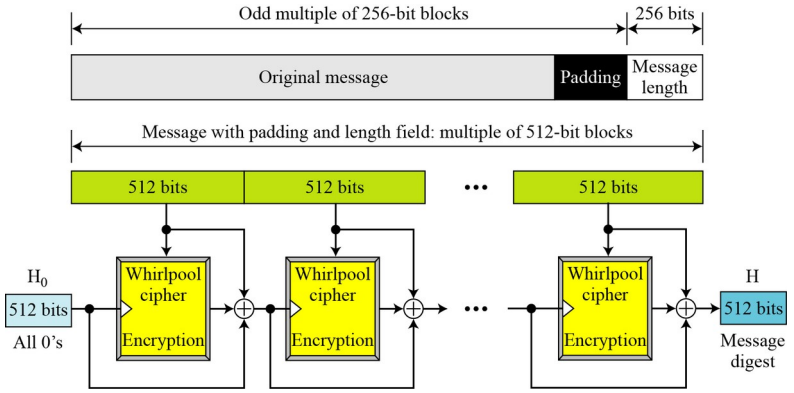
Endorsed by the New European Schemes for Signatures, Integrity, and Encryption (NESSIE).

12.3 WHIRLPOOL

Figure 12.12 ***Whirlpool hash function***



12.3 WHIRLPOOL



Preparation

Whirlpool requires that the length of the original message be less than 2^{256} bits.

A message needs to be padded before being processed.

The padding is a single 1-bit followed by the necessary numbers of 0-bits to make the length of the padding an odd multiple of 256 bits.

After padding, a block of 256 bits is added to define the length of the original message.

After padding and adding the length field, the augmented message size is an even multiple of 256 bits or a multiple of 512 bits.

12.3 WHIRLPOOL

Whirlpool creates a digest of 512 bits from a multiple 512-bit block message.

The 512-bit digest, H_0, 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.

Message digest is the final 512-bit ciphertext after the last exclusive-or operation.

Whirlpool Cipher

The Whirlpool cipher is a non-Feistel cipher like AES.

Here the Whirlpool cipher is compared with the AES cipher and their differences are mentioned.

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, K_0 to K_{10}, each of 512 bits.

Figure 12.13 *General idea of the Whirlpool cipher*

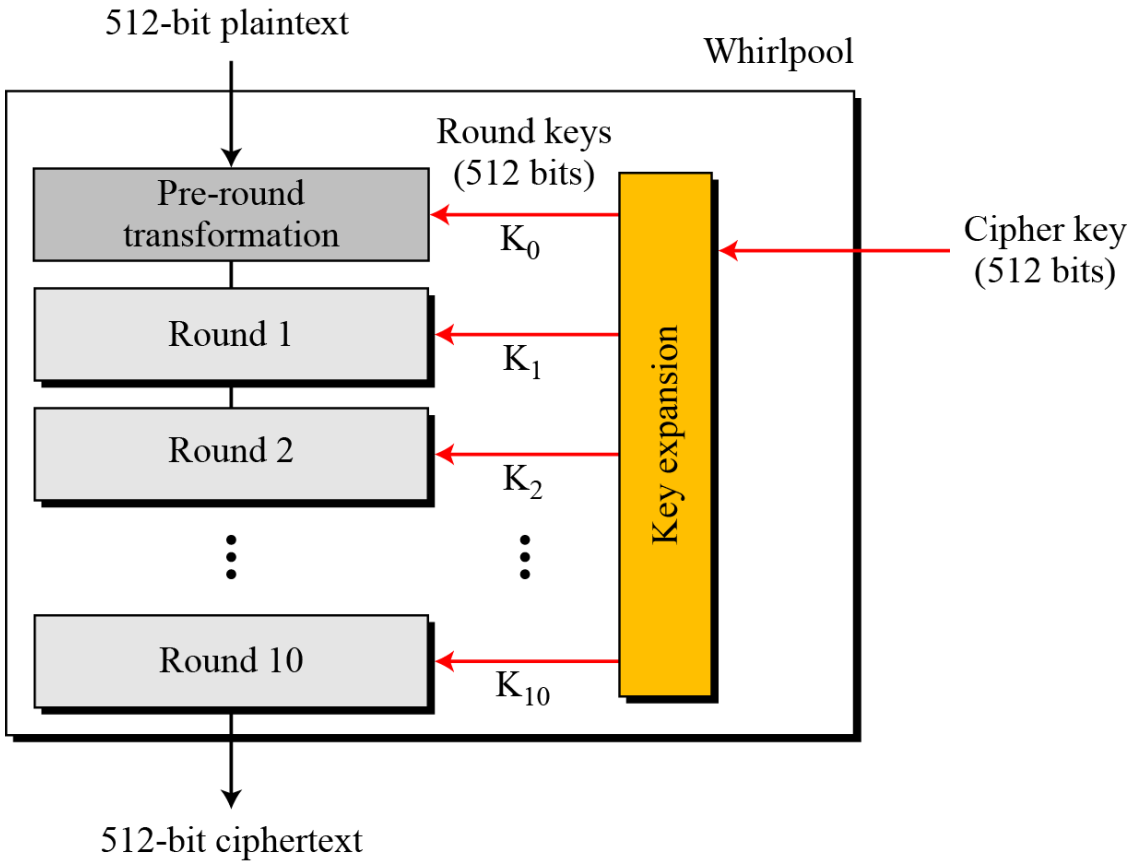


Figure 12.13 shows the general design of the Whirlpool cipher.

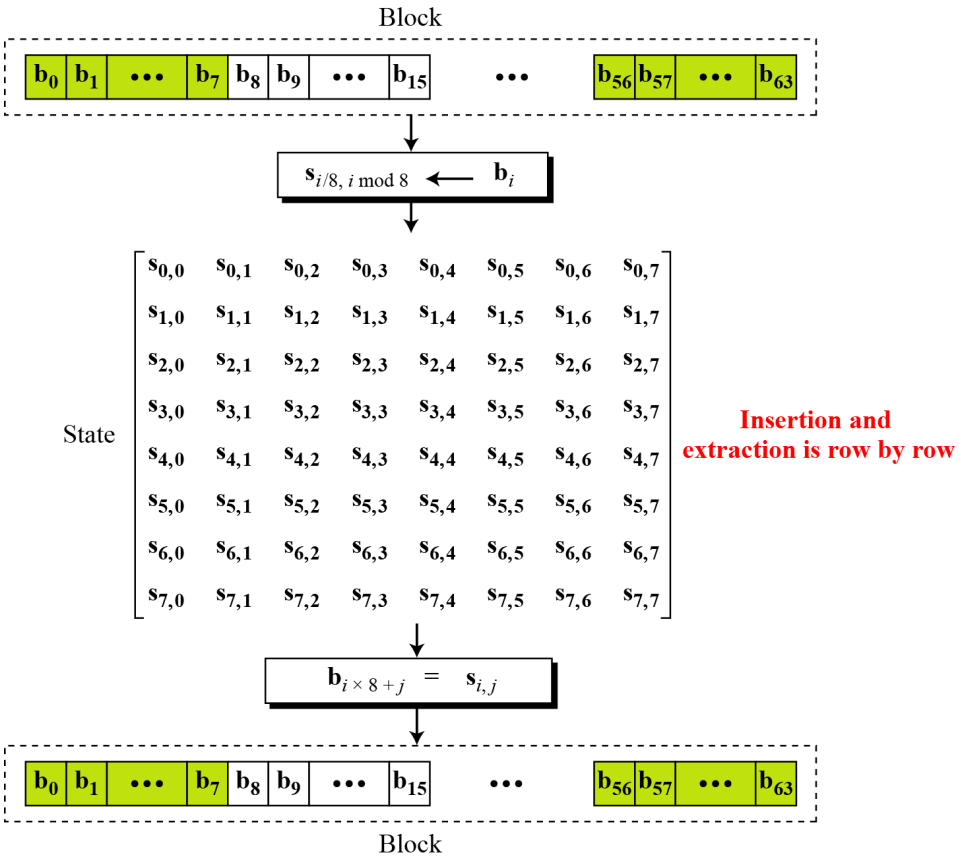
States and Blocks

Like the AES cipher, the Whirlpool cipher uses states and blocks.

However, the 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.

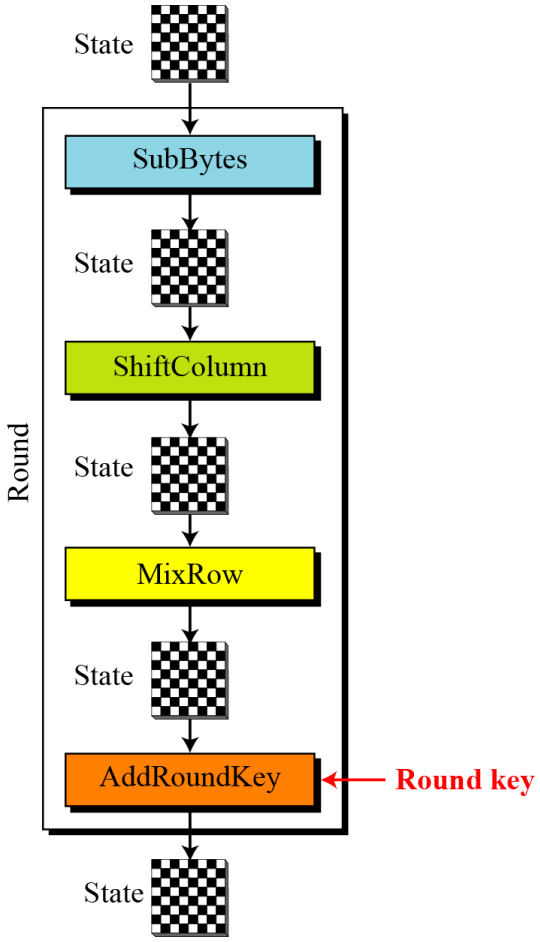
Figure 12.14 **Block and state in the Whirlpool cipher**



Unlike AES, the block-to-state or stateto- block transformation is done row by row.

Structure of Each Round

Figure 12.15 shows the structure of each round. Each round uses four transformations.



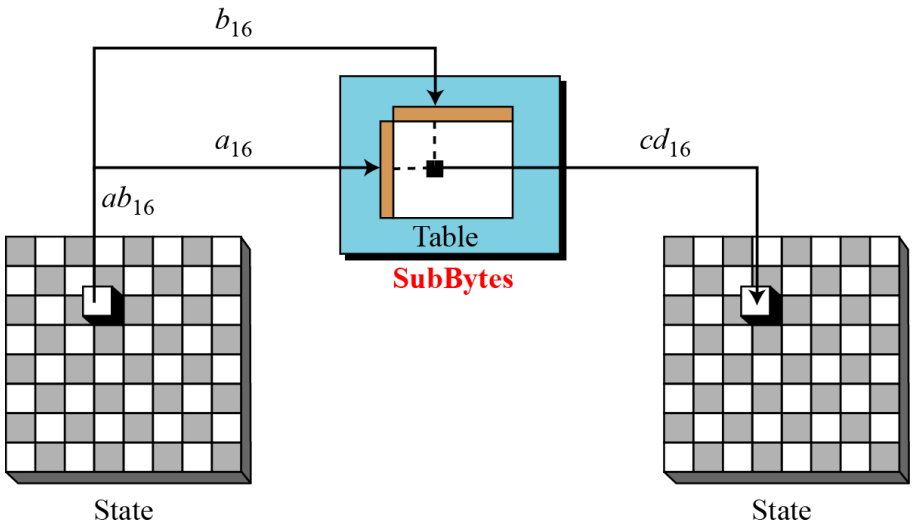
SubBytes Like in AES, SubBytes provide a nonlinear transformation.

A byte is represented as two hexadecimal digits.

Left digit defines the row and right digit defines the column of the substitution table.

Two hexadecimal digits at the junction of the row and the column are the new byte.

Figure 12.16 *SubBytes transformations in the Whirlpool cipher*



In the SubBytes transformation, the state is treated as an 8 × 8 matrix of bytes.

Transformation is done one byte at a time.

The contents of each byte are changed, but the arrangement of the bytes in the matrix remains the same.

Table 12.4 shows the substitution table (S-Box) for SubBytes transformation.

The transformation definitely provides confusion effect.

Table 12.4 *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

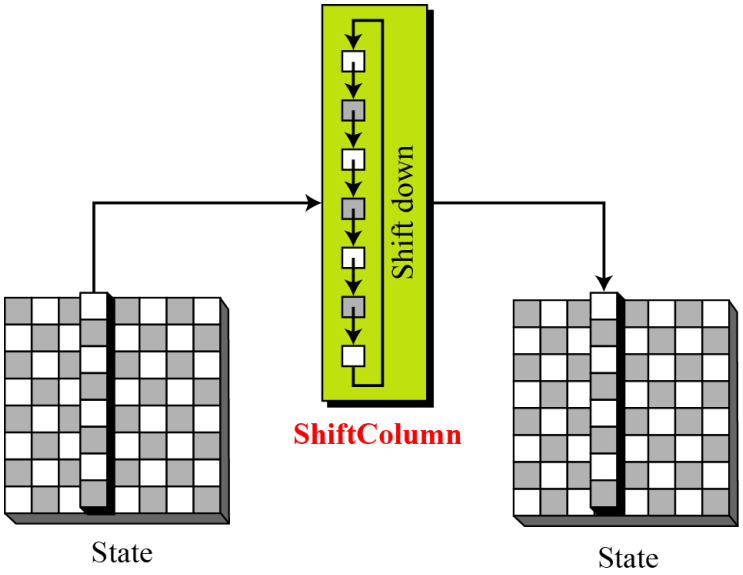
For example, two bytes, $5A_{16}$ and $5B_{16}$, which differ only in one bit (the rightmost bit), are transformed to $5B_{16}$ and 88_{16}, which differ in five bits.

ShiftColumns

To provide permutation, Whirlpool uses the ShiftColumns transformation, which is 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

The MixRows transformation has the same effect as the MixColumns transformation in AES: it diffuses the bits.

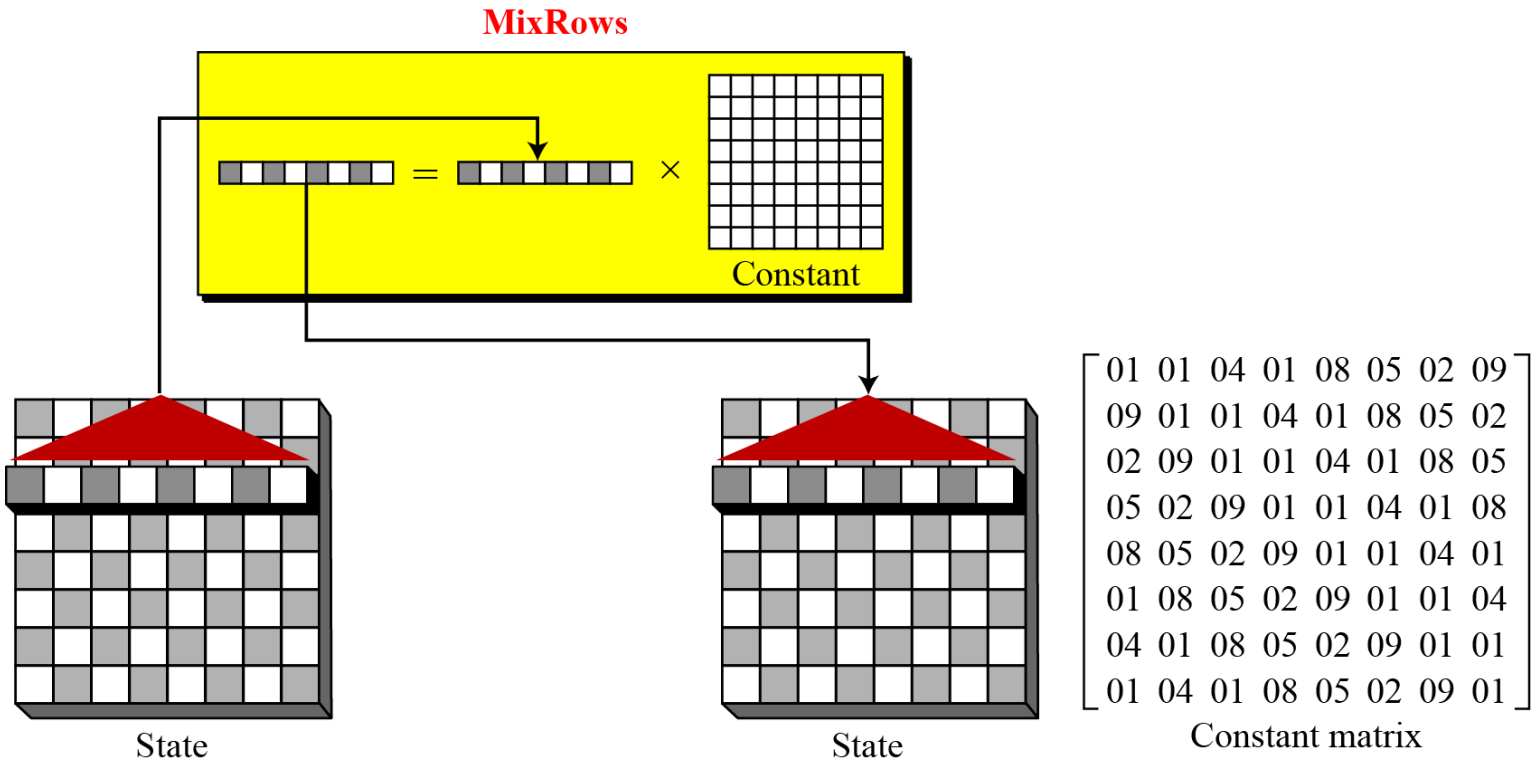
The MixRows transformation is a matrix transformation where bytes are interpreted as 8-bit words (or polynomials) with coefficients in GF(2).

Multiplication of bytes is done in $GF(2^8)$, but the modulus is different from the one used in AES.

The Whirlpool cipher uses (0x11D) or $(x^8 + x^4 + x^3 + x^2 + 1)$ as the modulus.

Addition is the same as XORing of 8-bit words.

Figure 12.19 *MixRows transformation in the Whirlpool cipher*



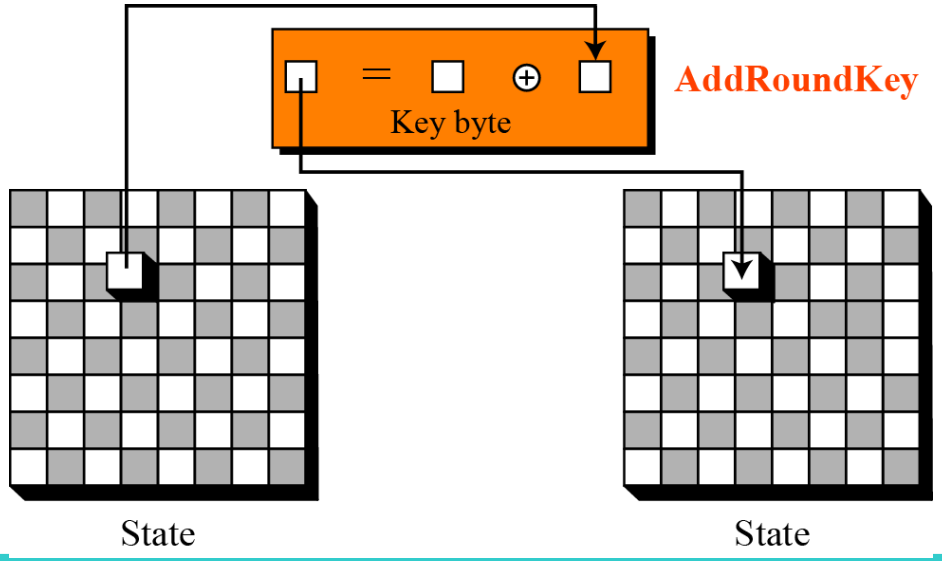
AddRoundKey

The AddRoundKey transformation in the Whirlpool cipher is done byte by byte, because each round key is also a state of an 8 × 8 matrix.

A byte from the data state is added, in GF(2^8) field, to the corresponding byte in the round-key state.

The result is the new byte in the new state.

Figure 12.20 ***AddRoundKey transformation in the Whirlpool cipher***



Key Expansion

Key-expansion algorithm in Whirlpool is totally different from the algorithm in AES.

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.

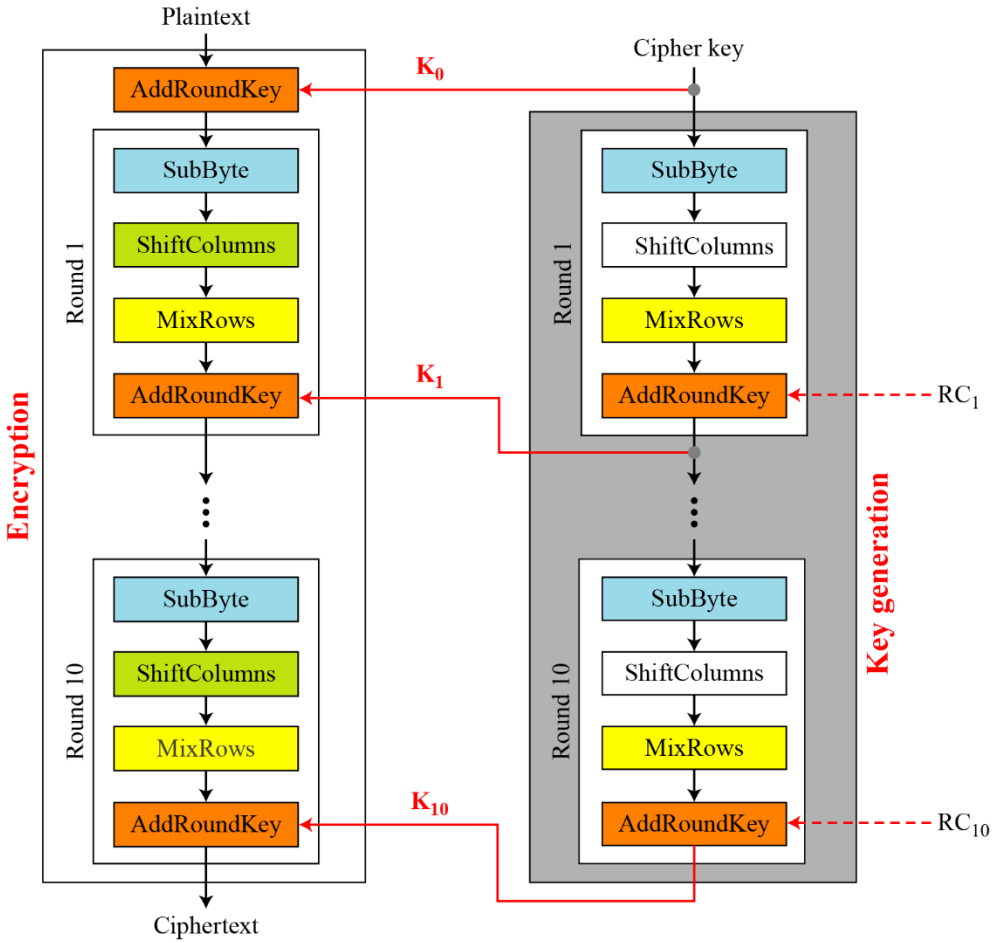
The output of each round in the encryption algorithm is the round key for that round.

Whirlpool uses ten round constants (RCs) as the virtual round keys for the keyexpansion algorithm.

In other words, the key-expansion algorithm uses constants as the round keys and the encryption algorithm uses the output of each round of the keyexpansion 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 K_0 for the encryption algorithm


Plaintext
Cipher key
AddRoundKey
K0
SubByte
ShiftColumns
MixRows
AddRoundKey
K1
RC1
Round 1
Encryption
Key generation
SubByte
ShiftColumns
MixRows
AddRoundKey
K10
RC10
Round 10
Ciphertext

Round Constants Each round constant, RC_r 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 (Table 12.4).

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

In other words, RC_1 uses the first eight entries in the SubBytes transformation table [Table 12.4]; RC_2 uses the second eight entries, and so on.

For example, Figure 12.22 shows RC_3, where the first row is the third eight entries in the SubBytes table.

Figure 12.22 *Round constant for the third round*

$$RC_3 = \begin{bmatrix} 1D & E0 & D7 & C2 & 2E & 4B & FE & 57 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \end{bmatrix}$$

Summary

Table 12.5 summarizes some characteristics of the Whirlpool cipher.

Table 12.5 *Main characteristics of the Whirlpool cipher*

Block size: 512 bits
Cipher key size: 512 bits
Number of rounds: 10
Key expansion: using the cipher itself with round constants as round keys
Substitution: SubBytes transformation
Permutation: ShiftColumns transformation
Mixing: MixRows transformation
Round Constant: cubic roots of the first eighty prime numbers

Analysis

Although Whirlpool has not been extensively studied or tested, it is based on a robust scheme (Miyaguchi-Preneel), and for a compression function uses a cipher that is based on AES, a cryptosystem that has been proved very resistant to attacks.

In addition, the size of the message digest is the same as for SHA-512.

Therefore it is expected to be a very strong cryptographic hash function.

However, more testing and researches are needed to confirm this.

The only concern is that Whirlpool, which is based on a cipher as the compression function, may not be as efficient as SHA-512, particularly when it is implemented in hardware.

END