CRYPTOGRAPHIC HASH FUNCTIONS

Mukesh Chinta

ASST PROFESSOR, CSE

V R S E C

- A hash function H accepts a variable-length block of data as input and produces a fixed-size hash value h = H(M) with the principal objective of maintaining Data Integrity.

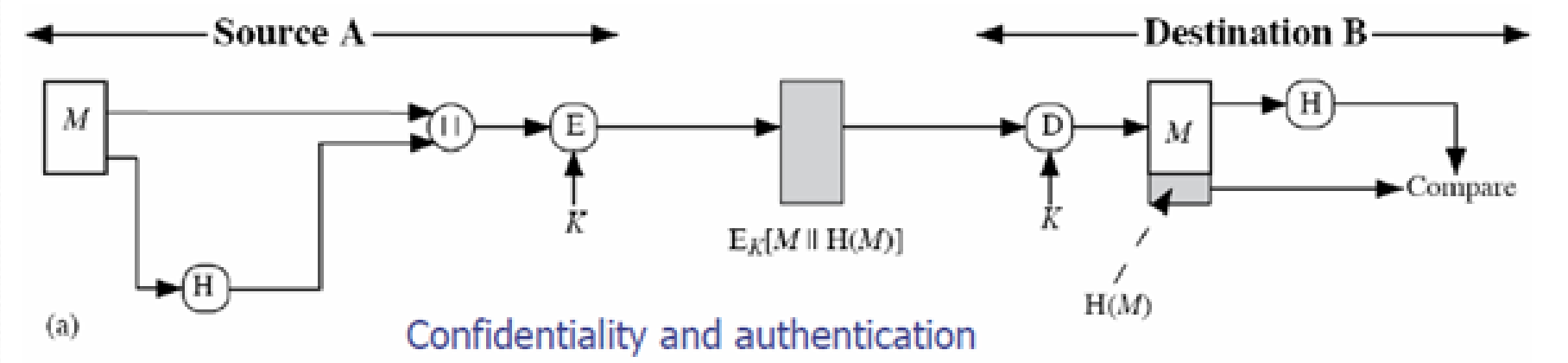
- The kind of hash function needed for security applications is referred to as a cryptographic hash function.

- A cryptographic hash function is an algorithm for which it is computationally infeasible to find either
  - (a) a data object that maps to a pre-specified hash result (**the one-way property**)
  - (b) two data objects that map to the same hash result (**the collision-free property**).

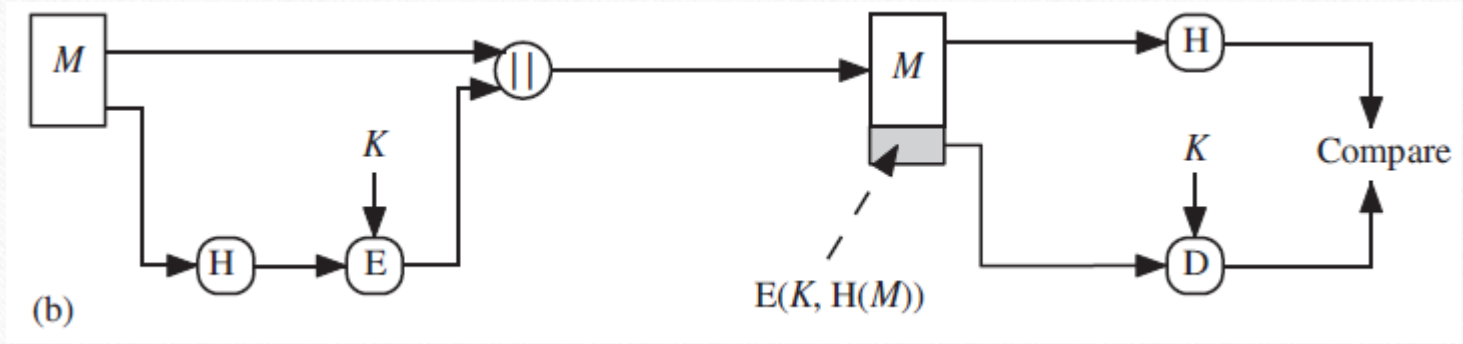
Because of these characteristics, hash functions are often used to determine whether or not data has changed

Message Authentication

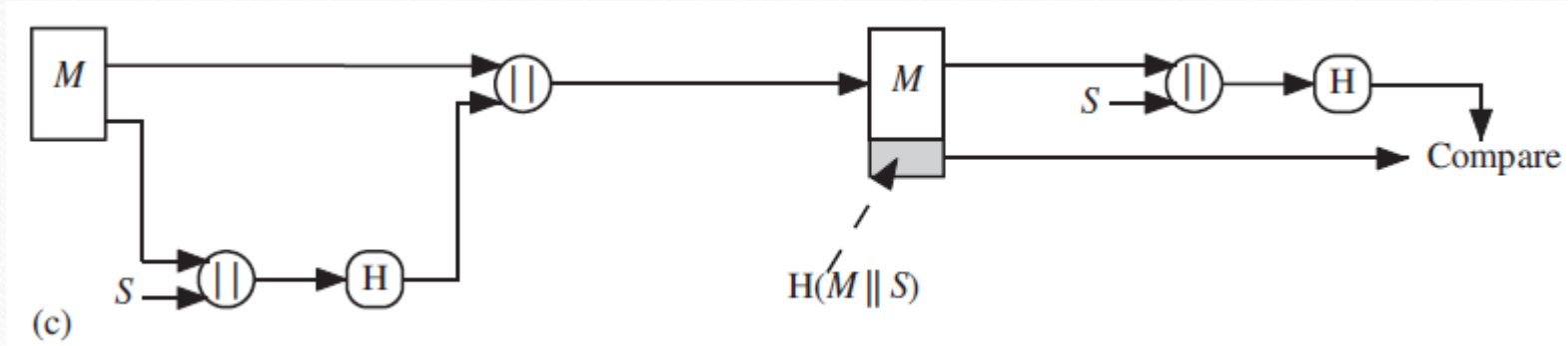
- Message authentication is a mechanism or service used to verify the integrity of a message. Message authentication assures that data received are exactly as sent (i.e., contain no modification, insertion, deletion, or replay).
- When a hash function is used to provide message authentication, the hash function value is often referred to as a message digest.
- A hash code can be used to provide message authentication in many ways.



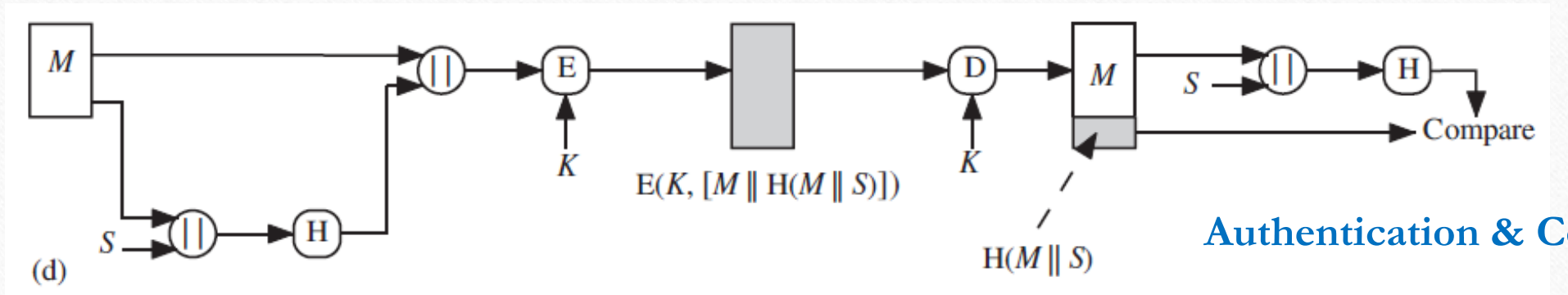
Confidentiality and authentication



Authentication



Authentication without encryption!!!

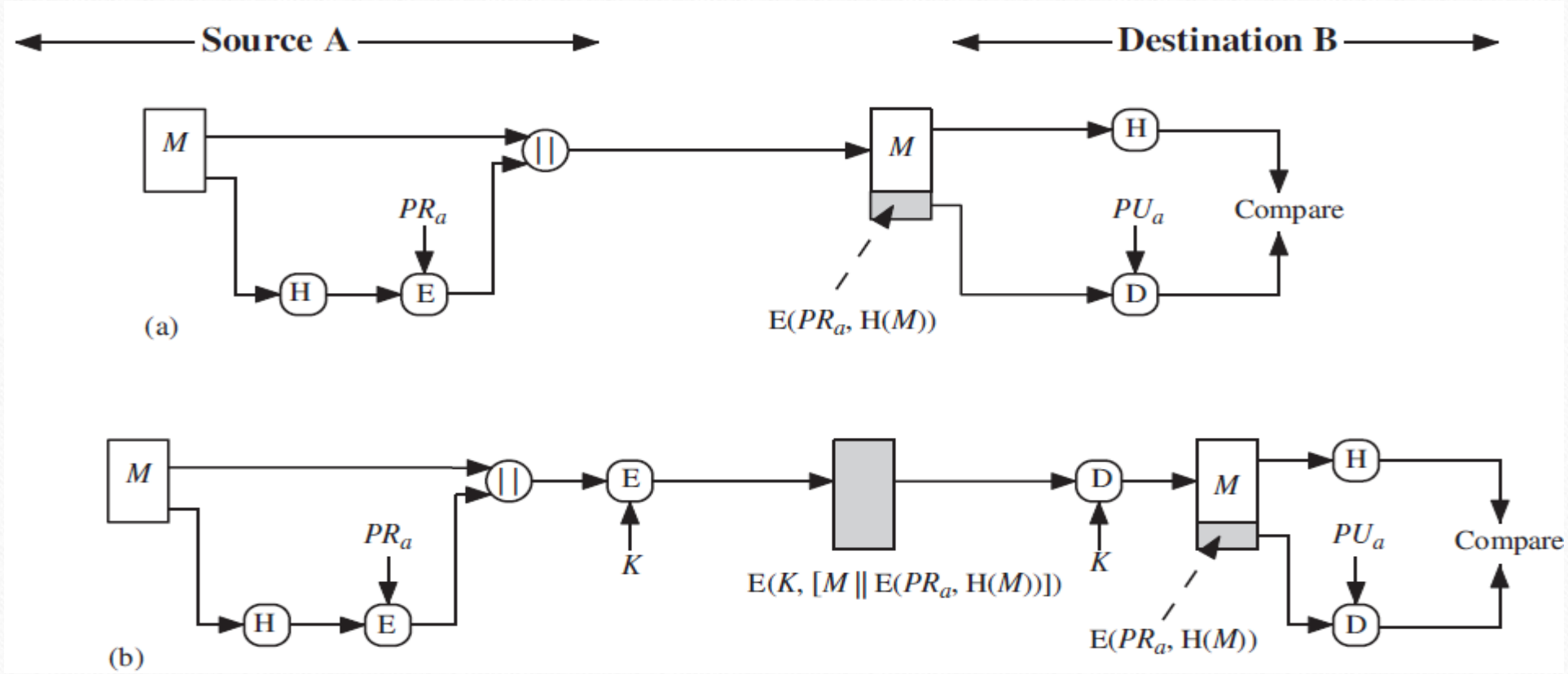


Authentication & Confidentiality

- Message authentication is achieved using a **message authentication code (MAC)**, also known as a **keyed hash** function.
- Typically, MACs are used between two parties that share a secret key to authenticate information exchanged between those parties.
- A MAC function takes as input a secret key and a data block and produces a hash value, referred to as the MAC. This can then be transmitted with or stored with the protected message.
- If the integrity of the message needs to be checked, the MAC function can be applied to the message and the result compared with the stored MAC value. An attacker who alters the message will be unable to alter the MAC value without knowledge of the secret key. The verifying party also knows who the sending party is because no one else knows the secret key.
- **E(K,H(M))** is a function of a variable-length message and a secret key , and it produces a fixed-size output that is secure against an opponent who does not know the secret key.

Digital Signature

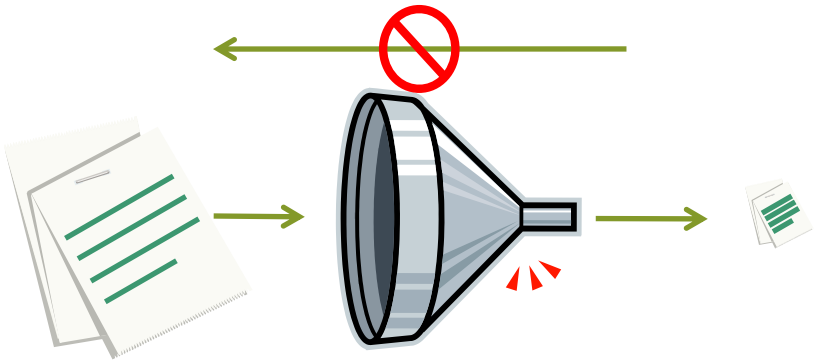
The operation of the digital signature is similar to that of the MAC. In the case of the digital signature, the hash value of a message is encrypted with a user's private key. Anyone who knows the user's public key can verify the integrity of the message that is associated with the digital signature.



REQUIREMENTS AND SECURITY

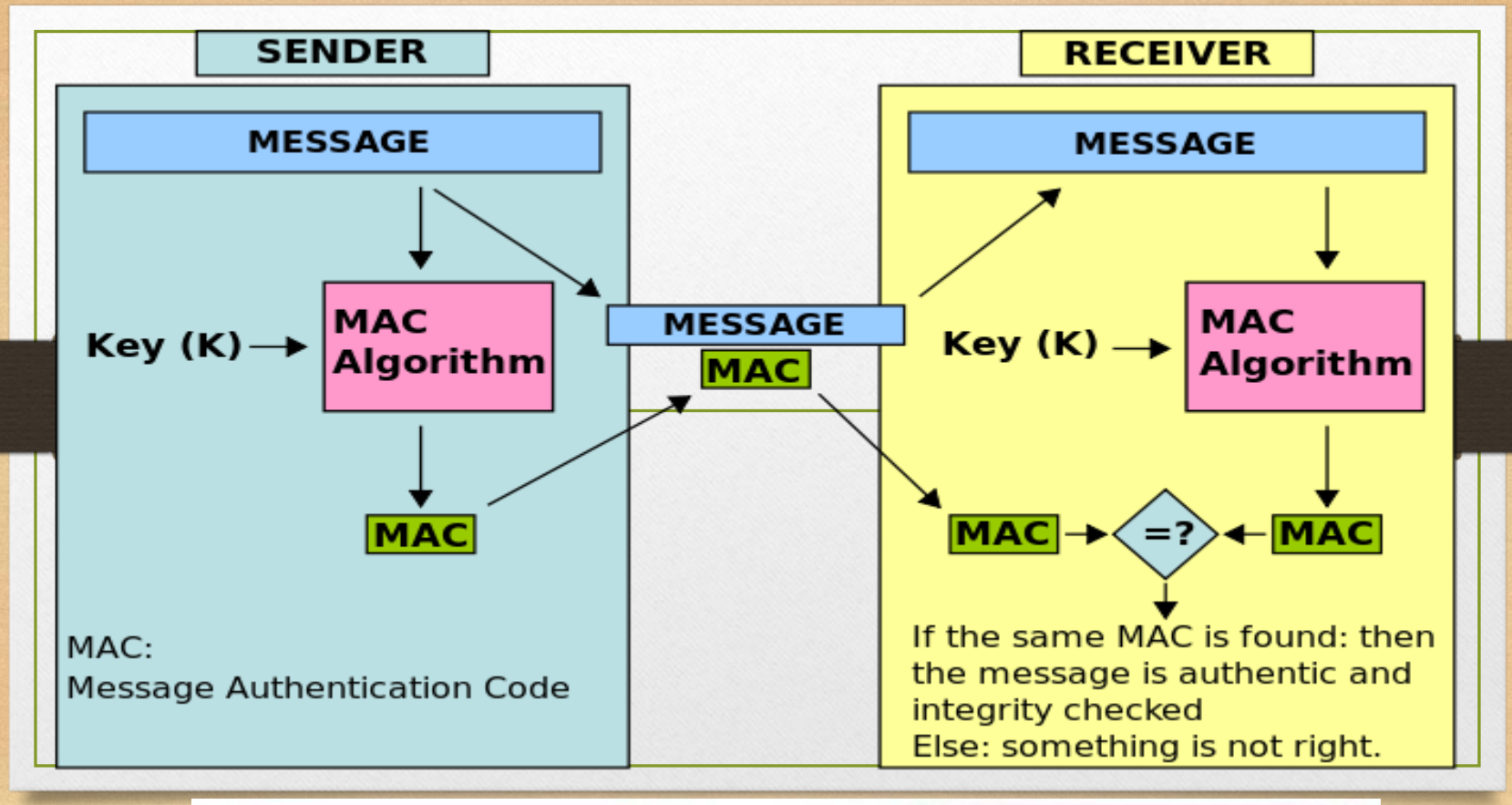
- A fixed-length hash value h is generated by a function H that takes as input a message of arbitrary length: **h=H(M).**
- For a hash value h = H(x), we say x that is the **preimage** of h. That is, x is a data block whose hash function, using the function H, is h.
- A **collision** occurs if we have x ≠ y and H(x) = H(y). As hash functions are used for data integrity, collisions are undesirable.

The purpose of a hash function is to produce a "fingerprint" of a file, message, or other block of data. To be used for message authentication, the hash function H must have the properties shown:



Requirement	Description
Variable input size	H can be applied to a block of data of any size.
Fixed output size	H produces a fixed-length output.
Efficiency	H(x) is relatively easy to compute for any given x, making both hardware and software implementations practical.
Preimage resistant (one-way property)	For any given hash value h, it is computationally infeasible to find y such that H(y) = h.
Second preimage resistant (weak collision resistant)	For any given block x, it is computationally infeasible to find $y \neq x$ with H(y) = H(x).
Collision resistant (strong collision resistant)	It is computationally infeasible to find any pair (x, y) such that H(x) = H(y).
Pseudorandomness	Output of H meets standard tests for pseudorandomness.

MESSAGE AUTHENTICATION CODES



A **Message authentication code (MAC)** is an algorithm that requires the use of a secret key. A MAC takes a variable-length message and a secret key as input and produces an authentication code. A recipient in possession of the secret key can generate an authentication code to verify the integrity of the message.

Following attacks can be identified in now a days communication:

1. Disclosure
2. Traffic Analysis
3. Masquerade
4. Content Modification
5. Sequence Modification
6. Timing Modification
7. Source Repudiation
8. Destination Repudiation

- Measures to deal with the first two attacks are in the realm of message confidentiality.
- Measures to deal with items (3) through (6) in the foregoing list are generally regarded as message authentication.
- Mechanisms for dealing specifically with item (7) come under the scope of digital signatures.

Message Authentication Functions

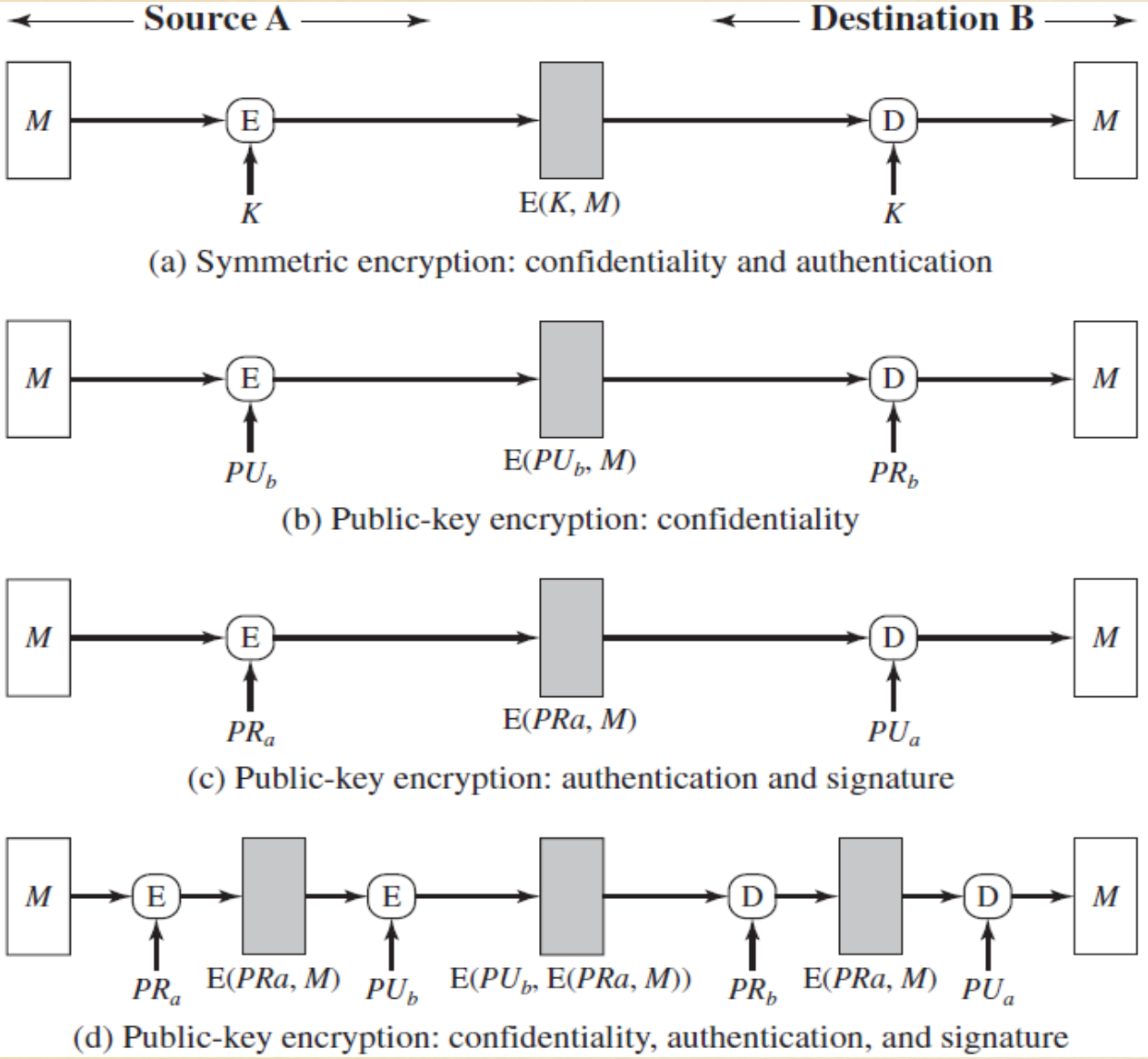
Message authentication is a procedure to verify that received messages come from the alleged source and have not been altered. Message authentication may also verify sequencing and timeliness. It is intended against the attacks like content modification, sequence modification, timing modification and repudiation.

There are three classes by which different types of functions that may be used to produce an authenticator.

- ***Message encryption***–the ciphertext serves as authenticator
- ***Message authentication code (MAC)***–a public function of the message and a secret key producing a fixed-length value to serve as authenticator. This does not provide a digital signature because A and B share the same key.
- ***Hash function***–a public function mapping an arbitrary length message into a fixed-length hash value to serve as authenticator. This does not provide a digital signature because there is no key.

Message Encryption

Message encryption by itself can provide a measure of authentication. The analysis differs for conventional and public-key encryption schemes



(a) Symmetric encryption: confidentiality and authentication

(b) Public-key encryption: confidentiality

(c) Public-key encryption: authentication and signature

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

A → B: $E_K[M]$
- Provides confidentiality
  - —Only A and B share K
- Provides a degree of authentication
  - —Could come only from A
  - —Has not been altered in transit
  - —Requires some formatting/redundancy
- Does not provide signature
  - —Receiver could forge message
  - —Sender could deny message

(a) Symmetric encryption

A → B: $E_{KU_b}[M]$
- Provides confidentiality
  - —Only B has KR_b to decrypt
- Provides no authentication
  - —Any party could use KU_b to encrypt message and claim to be A

(b) Public-key encryption: confidentiality

A → B: $E_{KR_a}[M]$
- Provides authentication and signature
  - —Only A has KR_a to encrypt
  - —Has not been altered in transit
  - —Requires some formatting/redundancy
  - —Any party can use KU_a to verify signature

(c) Public-key encryption: authentication and signature

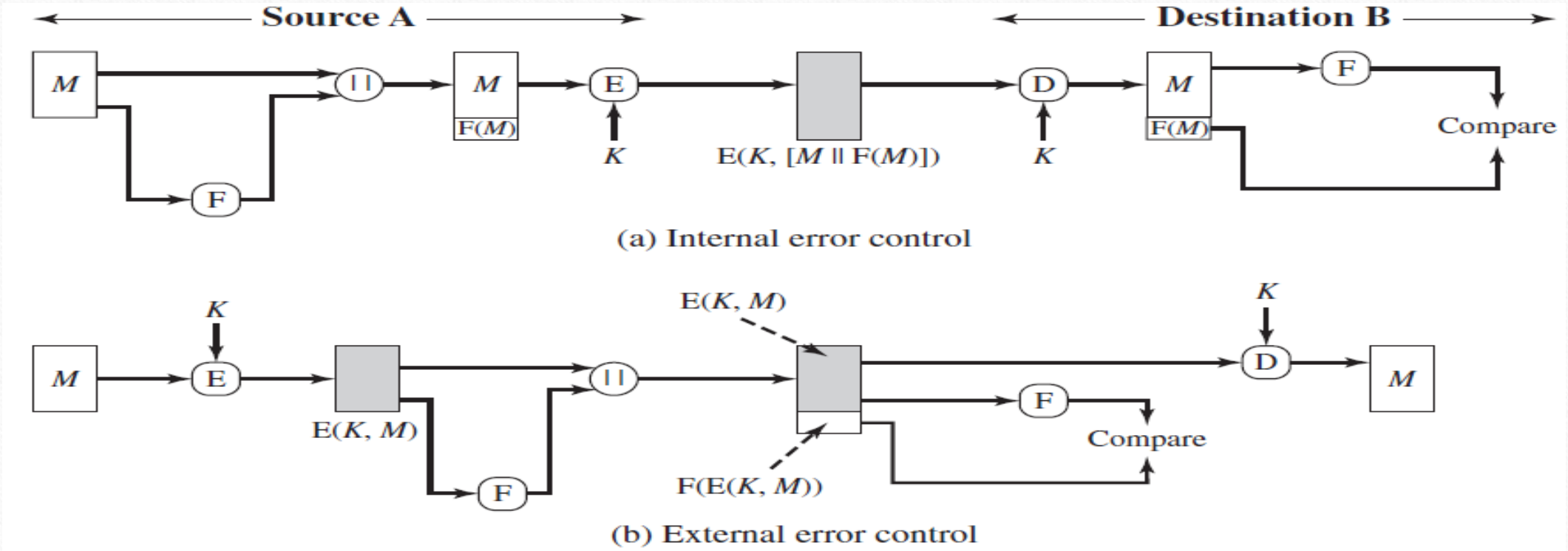
A → B: $E_{KU_b}[E_{KR_a}(M)]$
- Provides confidentiality because of KU_b
- Provides authentication and signature because of Kr_a

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

Often one needs alternative authentication schemes than just encrypting the message.

- Sometimes one needs to avoid encryption of full messages due to legal requirements.
- Encryption and authentication may be separated in the system architecture.

An opponent could achieve a certain level of disruption simply by issuing messages with random content purporting to come from a legitimate user. To counter such attacks, append an error-detecting code, also known as a frame check sequence (FCS) or checksum, to each message before encryption.



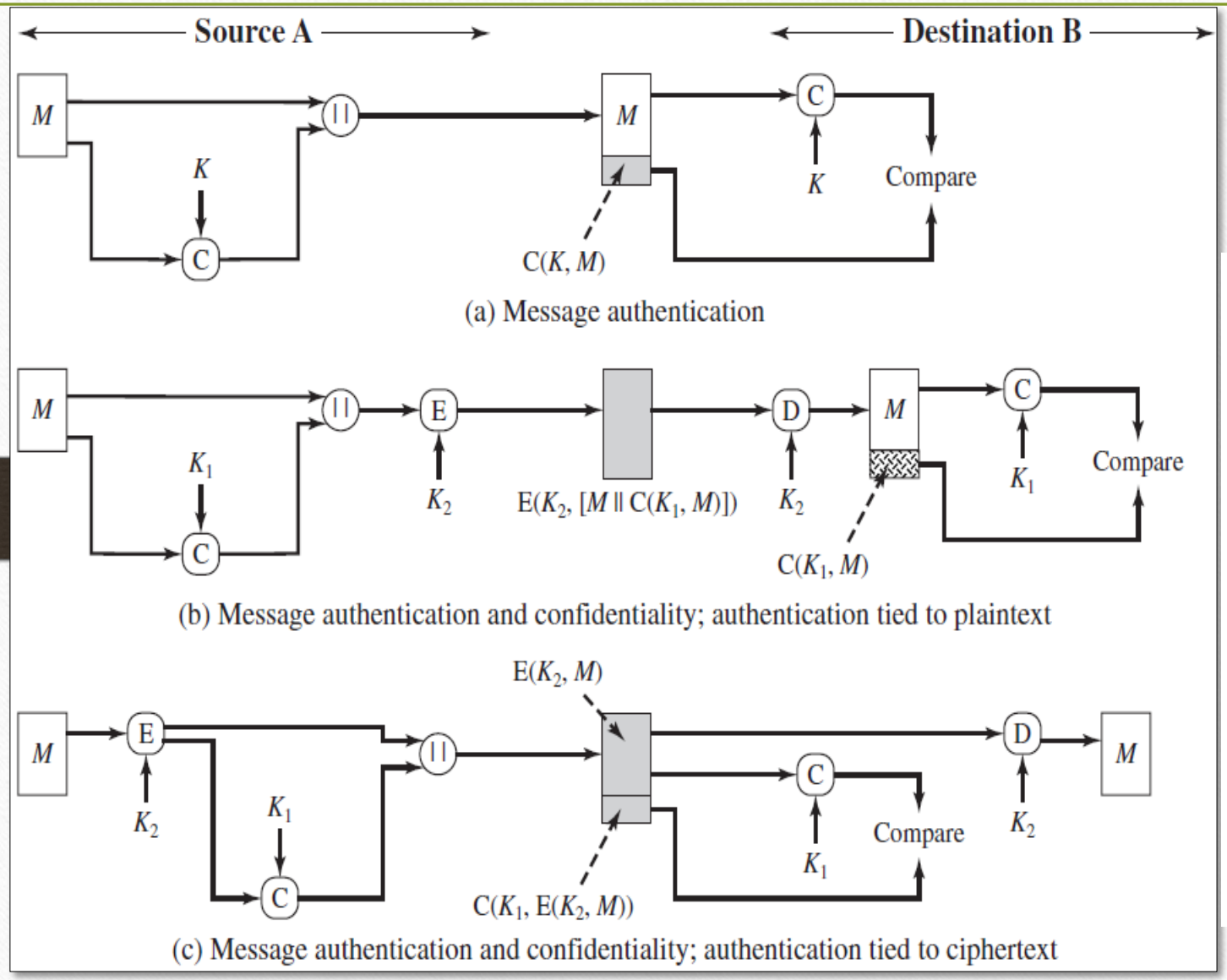
Message Authentication Code

An alternative authentication technique involves the use of a secret key to generate a small fixed-size block of data, known as cryptographic checksum or MAC, which is appended to the message. This technique assumes that both the communicating parties say A and B share a common secret key K. When A has a message to send to B, it calculates MAC as a function C of key and message given as: $\mathbf{MAC=C_k(M)}$

The message and the MAC are transmitted to the intended recipient, who upon receiving performs the same calculation on the received message, using the same secret key to generate a new MAC. The received MAC is compared to the calculated MAC and only if they match, then:

- The receiver is assured that the message has not been altered: Any alternations been done the MAC's do not match.
- The receiver is assured that the message is from the alleged sender: No one except the sender has the secret key and could prepare a message with a proper MAC.
- If the message includes a sequence number, then receiver is assured of proper sequence as an attacker cannot successfully alter the sequence number

Basic Uses of a Message Authentication Code



(a) $A \rightarrow B: M \| C_K(M)$

- Provides authentication
  - —Only A and B share K

(b) $A \rightarrow B: E_{K_2}[M \| C_{K_1}(M)]$

- Provides authentication
  - —Only A and B share K_1
- Provides confidentiality
  - —Only A and B share K_2

(c) $A \rightarrow B: E_{K_2}[M] \| C_{K_1}(E_{K_2}[M])$

- Provides authentication
  - —Using K_1
- Provides confidentiality
  - —Using K_2

Scenarios for MAC Usage

- A MAC function is similar to encryption. One difference is that the MAC algorithm need not be reversible, as it must be for decryption. In general, the MAC function is a many-to-one function.
- It turns out that, because of the mathematical properties of the authentication function, it is less vulnerable to being broken than encryption

- If a message is broadcast to several destinations in a network (such as a military control center), then it is cheaper and more reliable to have just one node responsible to evaluate the authenticity –message will be sent in plain with an attached authenticator.
- If one side has a heavy load, it cannot afford to decrypt all messages –it will just check the authenticity of some randomly selected messages.
- Authentication of computer programs in plaintext is very attractive service as they need not be decrypted every time wasting of processor resources. Integrity of the program can always be checked by MAC.
- For some applications, it may not be of concern to keep messages secret, but it is important to authenticate messages. An example is the Simple Network Management Protocol Version 3 (SNMPv3).
- Separation of authentication and confidentiality functions affords architectural flexibility.
- A user may wish to prolong the period of protection beyond the time of reception and yet allow processing of message contents

Requirements of a MAC Function

- If an opponent observes M and MAC(K,M), it should be computationally infeasible for the opponent to construct a message M' such that

MAC(K, M') = MAC(K, M)

- MAC(K,M) should be uniformly distributed in the sense that for randomly chosen messages, M and M', the probability that is MAC(K,M) = MAC(K,M') is 2^{-n}, where n is the number of bits in the tag.

- Let M' be equal to some known transformation on M. That is M' = f(M),. For example, f may involve inverting one or more specific bits. In that case,

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

- *The first requirement states a scenario, in which an opponent is able to construct a new message to match a given tag, even though the opponent does not know and does not learn the key.*
- *The second requirement deals with the need to thwart a brute-force attack based on chosen plaintext.*
- *The final requirement dictates that the authentication algorithm should not be weaker with respect to certain parts or bits of the message than others*

Limitations of a MAC Function

There are two major limitations of MAC, both due to its symmetric nature of operation −

Establishment of Shared Secret.

- It can provide message authentication among pre-decided legitimate users who have shared key. This requires establishment of shared secret prior to use of MAC.

Inability to Provide Non-Repudiation

- Non-repudiation is the assurance that a message originator cannot deny any previously sent messages and commitments or actions.
- MAC technique does not provide a non-repudiation service. If the sender and receiver get involved in a dispute over message origination, MACs cannot provide a proof that a message was indeed sent by the sender.
- Though no third party can compute the MAC, still sender could deny having sent the message and claim that the receiver forged it, as it is impossible to determine which of the two parties computed the MAC.

Message-Digest algorithms characteristics

Message-Digest (Fingerprint) algorithms are special functions which transform input of (usually) arbitrary length into output (so-called "fingerprint" or "message digest") of constant length. These transformation functions must fulfil these requirements:

- no one should be able to produce two different inputs for which the transformation function returns the same output
- no one should be able to produce input for given prespecified output

MD2, MD4 and MD5 are message-digest algorithms developed by Ronald L. Rivest. All these three algorithms take input message of arbitrary length and produce a 128-bit message digest.

MD2 - developed in 1989 (description available in Internet RFC 1319)

The message is first padded so its length in bytes is divisible by 16. A 16-byte checksum is then appended to the message, and the hash value is computed on the resulting message. MD2 was optimized for 8-bit machines, whereas MD4 and MD5 were aimed at 32-bit machines.

MD4 - developed in 1990 (description available in Internet RFC 1320)

The message is padded to ensure that its length in bits plus 64 is divisible by 512. A 64-bit binary representation of the original length of the message is then concatenated to the message. The message is processed in 512-bit blocks in the Damgard/Merkle iterative structure, and each block is processed in three distinct rounds.

MD5 - developed in 1991 (description available in Internet RFC 1321)

It is basically MD4 with "safety-belts" and while it is slightly slower than MD4, it is more secure. The algorithm consists of four distinct rounds, which has a slightly different design from that of MD4. Message-digest size, as well as padding requirements, remain the same.

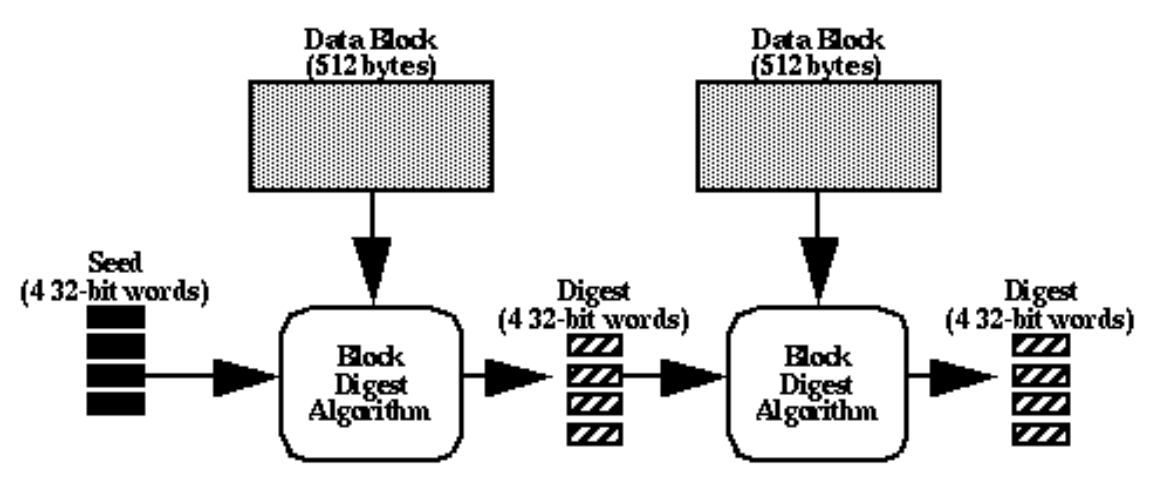
MD5 Message Digest Algorithm

The MD5 message-digest algorithm takes as input, a message of arbitrary length and produces as output, a 128-bit message digest. The input is processed in 512-bit blocks. The processing consists of the following steps:

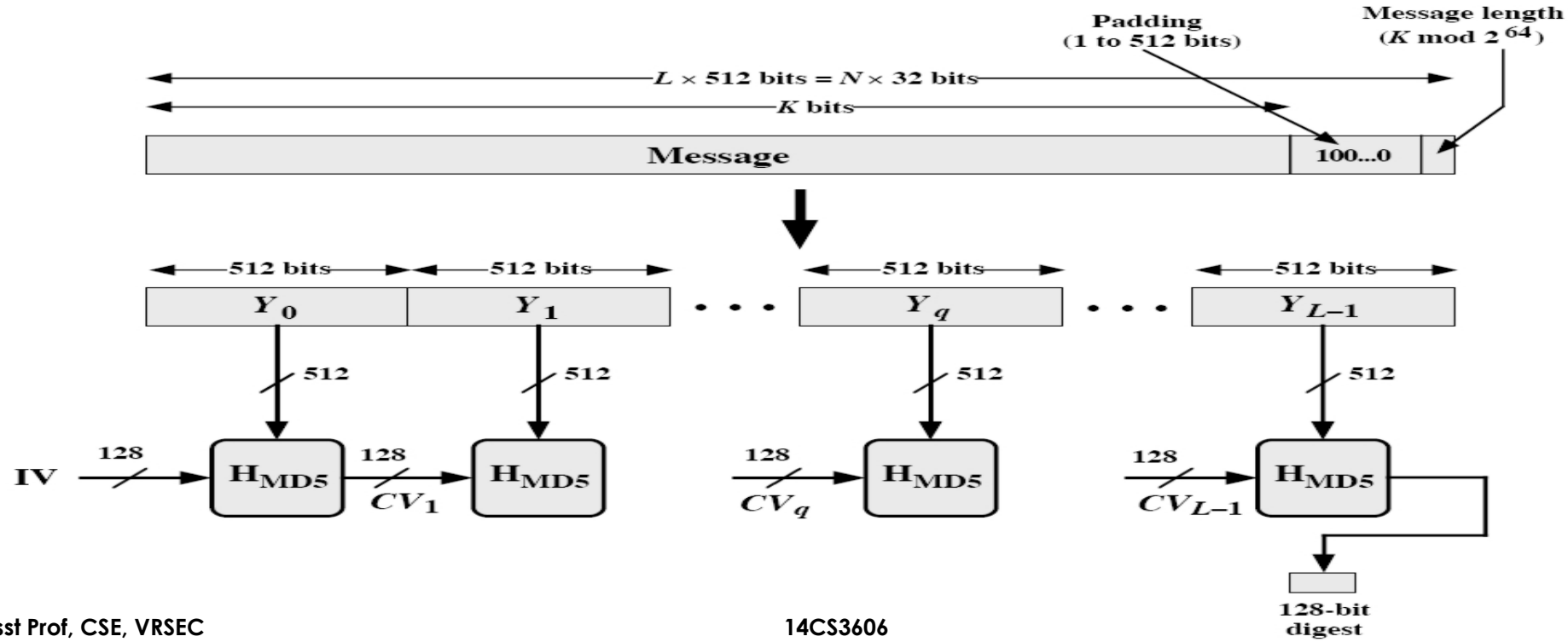
1. ***Append Padding bits***: The message is padded so that its length in bits is congruent to 448 modulo 512 i.e. the length of the padded message is 64 bits less than an integer multiple of 512 bits. Padding is always added, even if the message is already of the desired length. Padding consists of a single 1-bit followed by the necessary number of 0-bits.

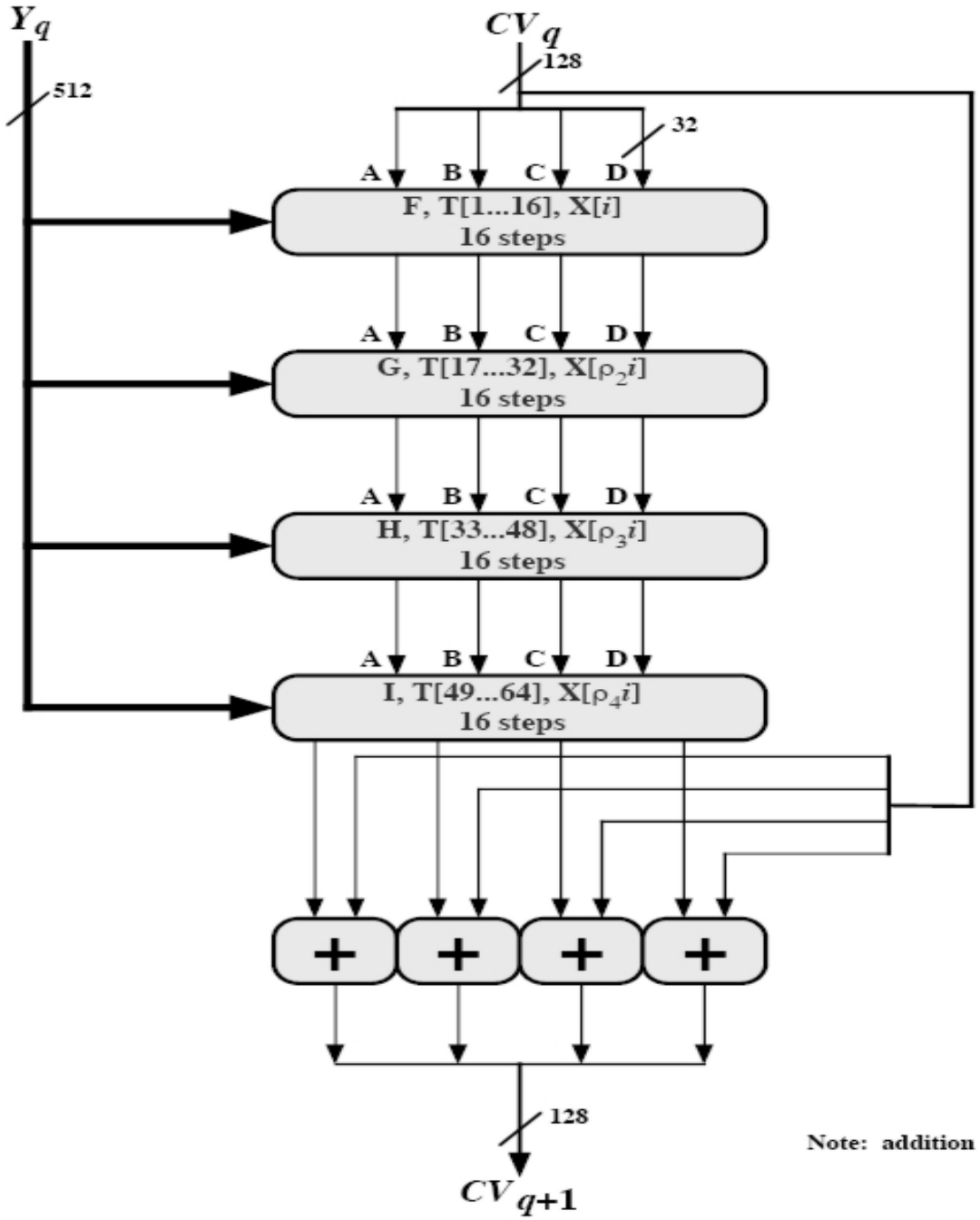
2. ***Append length***: A 64-bit representation of the length in bits of the original message (before the padding) is appended to the result of step-1. If the length is larger than 2^{64}, the 64 least representative bits are taken.

3. ***Initialize MD buffer***: A 128-bit buffer is used to hold intermediate and final results of the hash function. The buffer can be represented as four 32-bit registers (A, B, C, D) and are initialized with A=0x01234567, B=0x89ABCDEF, C=0xFEDCBA98, D=0x76543210 i.e. 32-bit integers (hexadecimal values).



4. ***Process Message in 512-bit (16-word) blocks:*** The heart of algorithm is the compression function that consists of four rounds of processing and this module is labeled H_{MD5}. The four rounds have a similar structure, but each uses a different primitive logical function, referred to as F, G, H and I in the specification. Each block takes as input the current 512-bit block being processed Yq and the 128-bit buffer value ABCD and updates the contents of the buffer. Each round also makes use of one-fourth of a 64-element table T[1….64], constructed from the sine function. The ith element of T, denoted ***T[i], has the value equal to the integer part of 2^{32} * abs(sin(i)),*** where i is in radians. As the value of abs(sin(i)) is a value between 0 and 1, each element of T is an integer that can be represented in 32-bits and would eliminate any regularities in the input data. The output of fourth round is added to the input to the first round (CV_q) to produce CV_{q+1}. The addition is done independently for each of the four words in the buffer with each of the corresponding words in CV_q, using addition modulo 2^{32}.





5) Output: After all L, 512-bit blocks have been processed, the output from the L^{th} stage is the 128-bit message digest. MD5 can be summarized as follows:

$$CV0 = IV$$

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

$$MD = CV_L$$

Where,

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

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

L = the number of blocks in the message

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

RFx = round function using primitive logical function x.

MD = final message digest value

SUM_{32} = Addition modulo 2^{32} performed separately.

MD5 Compression Function

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

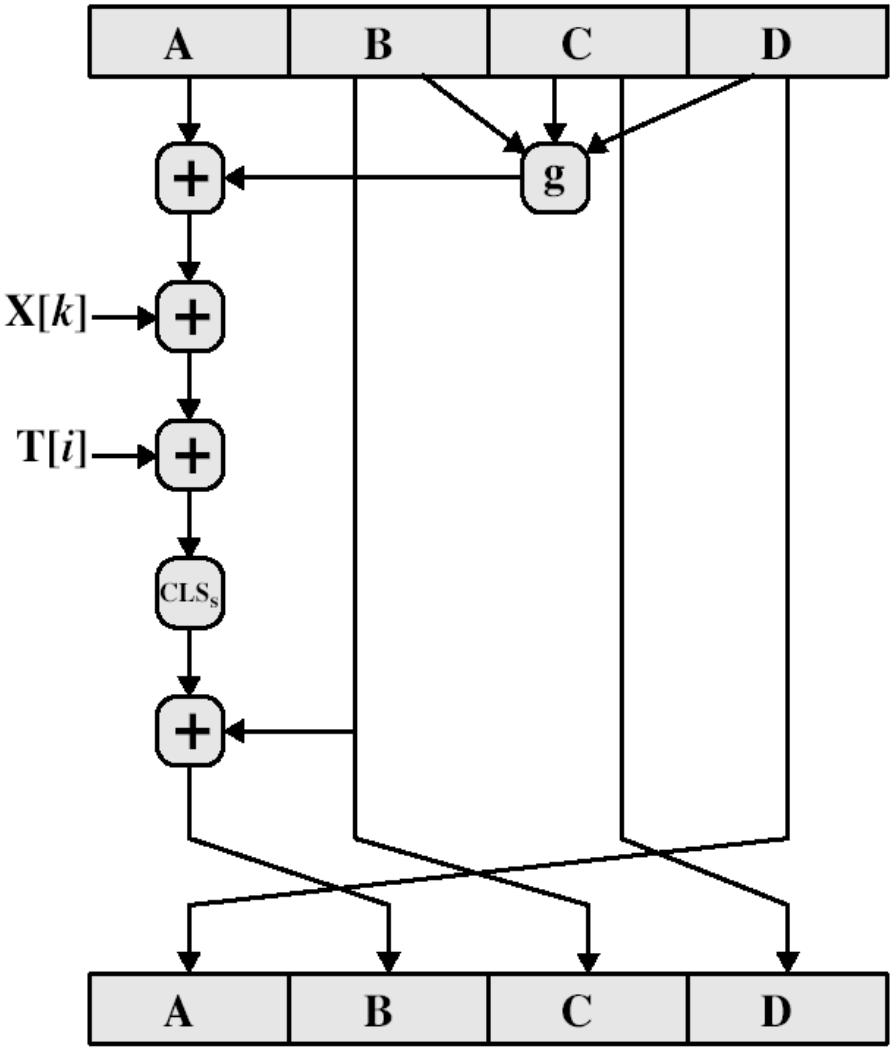
$$a = b+((a+g(b,c,d)+X[k]+T[i])<<<s)$$

where a, b, c, d refer to the four words of the buffer but used in varying permutations. After 16 steps, each word is updated 4 times. g(b,c,d) is a different nonlinear function in each round (F,G,H,I).

The primitive function g of the F,G,H,I is given as:

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

Where the logical operators (AND, OR, NOT, XOR) are represented by the symbols ($\wedge$, $\vee$, ~, $\oplus$)



MD5 operation of a single step

Each round mixes the buffer input with the next "word" of the message in a complex, non-linear manner. A different non-linear function is used in each of the 4 rounds (but the same function for all 16 steps in a round). The 4 buffer words (a,b,c,d) are rotated from step to step so all are used and updated. g is one of the primitive functions F,G,H,I for the 4 rounds respectively. X[k] is the kth 32-bit word in the current message block. T[i] is the i^{th} entry in the matrix of constants T. The addition of varying constants T and the use of different shifts helps ensure it is extremely difficult to compute collisions.

The array of 32-bit words X[0..15] holds the value of current 512-bit input block being processed. Within a round, each of the 16 words of X[i] is used exactly once, during one step. The order in which these words is used varies from round to round. In the first round, the words are used in their original order. For rounds 2 through 4, the following permutations are used

- $\rho_2(i) = (1 + 5i) \bmod 16$
- $\rho_3(i) = (5 + 3i) \bmod 16$
- $\rho_4(i) = 7i \bmod 16$

for i from 0 to 63 K[i] := floor(2^{32} × abs(sin(i + 1)))
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 }

```
//s specifies the per-round shift amounts
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 }
```

Security of MD5

- In 2004 it was shown that MD5 is not collision resistant. As such, MD5 is not suitable for applications like SSL certificates or digital signatures that rely on this property for digital security.
- Also in 2004 more serious flaws were discovered in MD5, making further use of the algorithm for security purposes questionable. Specifically, a group of researchers described how to create a pair of files that share the same MD5 checksum.
- Further advances were made in breaking MD5 in 2005, 2006, and 2007.
- In December 2008, a group of researchers used this technique to fake SSL certificate validity, and CMU Software Engineering Institute now says that **"MD5 should be considered cryptographically broken and unsuitable for further use"**

Secure Hash Algorithm

- The Secure Hash Algorithm (SHA) was developed by the National Institute of Standards and Technology (NIST) and published as a federal information processing standard (FIPS 180) in 1993; a revised version was issued as FIPS 180-1 in 1995 and is generally referred to as SHA-1.
- SHA-1 produces a hash value of 160 bits. In 2002, NIST produced a revised version of the standard, FIPS 180-2, that defined three new versions of SHA, with hash value lengths of 256, 384, and 512 bits, known as SHA-256, SHA-384, and SHA-512.

Table 11.3 Comparison of SHA Parameters

	SHA-1	SHA-224	SHA-256	SHA-384	SHA-512
Message Digest Size	160	224	256	384	512
Message Size	$< 2^{64}$	$< 2^{64}$	$< 2^{64}$	$< 2^{128}$	$< 2^{128}$
Block Size	512	512	512	1024	1024
Word Size	32	32	32	64	64
Number of Steps	80	64	64	80	80

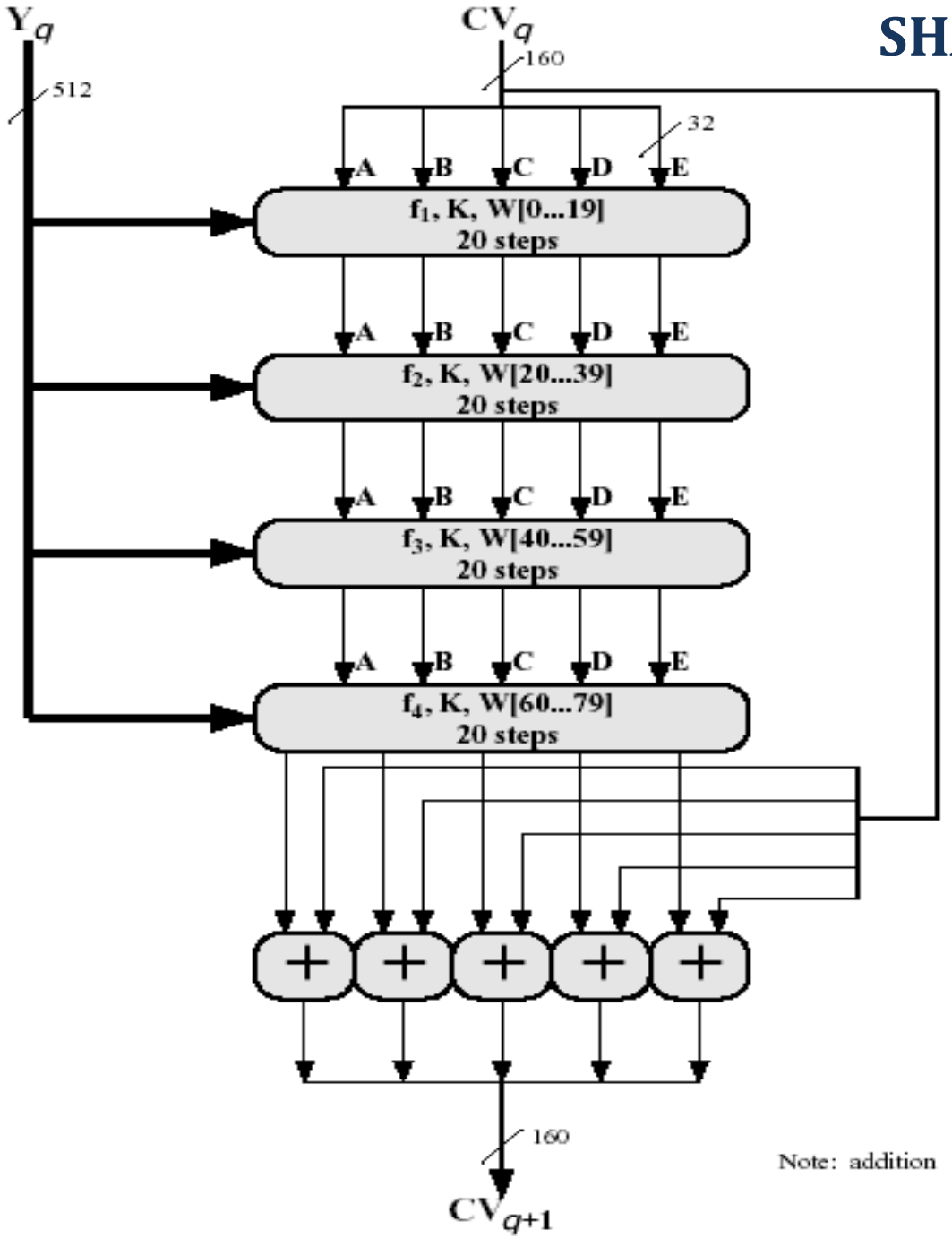
Note: All sizes are measured in bits.

- The algorithm takes as input a message with a maximum length of less than 2^{64} bits and produces as output a 160-bit message digest.
- The input is processed in 512-bit blocks. The overall processing of a message follows the structure of MD5 with block length of 512 bits and a hash length and chaining variable length of 160 bits.

The processing consists of following steps:

- **Append Padding Bits:** The message is padded so that length is congruent to 448 modulo 512; padding always added –one bit 1 followed by the necessary number of 0 bits.
- **Append Length**: a block of 64 bits containing the length of the original message is added.
- **Initialize MD buffer**: A 160-bit buffer is used to hold intermediate and final results on the hash function. This is formed by 32-bit registers A,B,C,D,E. Initial values: A=0x67452301, B=0xEFCDAB89, C=0x98BADCFE, D=0x10325476, E=C3D2E1F0. Stores in big-endian format i.e. the most significant bit in low address.
- **Process message in blocks 512-bit (16-word) blocks:** The processing of a single 512-bit block is shown. It consists of four rounds of processing of 20 steps each. These four rounds have similar structure, but uses a different primitive logical function, which we refer to as f1, f2, f3 and f4. Each round takes as input the current 512-bit block being processed and the 160-bit buffer value ABCDE and updates the contents of the buffer. Each round also makes use of four distinct additive constants Kt. The output of the fourth round i.e. eightieth step is added to the input to the first round to produce CV_{q+1}.

SHA-1 Processing of a single block



➢ **Output**: After all L 512-bit blocks have been processed, the output from the Lth stage is the 160-bit message digest.

The behavior of SHA-1 is as follows:

$\mathbf{CV_0 = IV}$

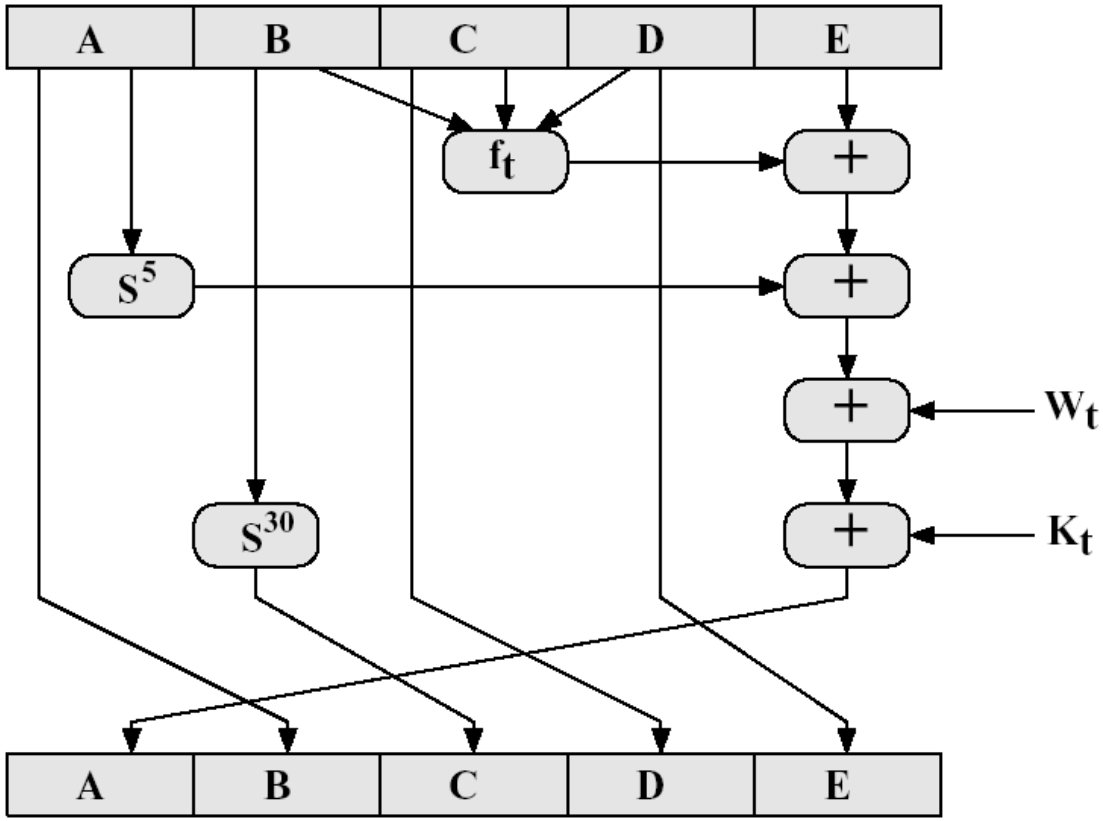
$\mathbf{CV_{q+1} = SUM_{32}(CV_q, ABCDE_q)}$

$\mathbf{MD = CV_L}$

Where,

- IV = initial value of ABCDE buffer
- $ABCDE_q$ = output of last round of processing of qth message block
- L = number of blocks in the message
- SUM_{32} = Addition modulo 2^{32}
- MD = final message digest value.

SHA-1 Compression Function



Each round has 20 steps which replaces the 5 buffer words.
The logic present in each one of the 80 rounds present is given as

(A,B,C,D,E) ⇐ (E + f(t,B,C,D) + S^5(A)+ Wt+ Kt), A, S^{30}(B), C, D

Where,

A, B, C, D, E = the five words of the buffer
t = step number; 0< t < 79
f(t,B,C,D) = primitive logical function for step t
Sk = circular left shift of the 32-bit argument by k bits
Wt = a 32-bit word derived from current 512-bit input block.
Kt = an additive constant; four distinct values are used
+ = modulo additon

SHA shares much in common with MD4/5, but with 20 instead of 16 steps in each of the 4 rounds. Note the 4 constants are based on sqrt(2,3,5,10).

Step Number	Hexadecimal	Ineteger Part of
$0 \le t \le 19$	K_t = 5A827999	$[2^{30} \times \sqrt{2}]$
$20 \le t \le 39$	K_t = 6ED9EBA1	$[2^{30} \times \sqrt{3}]$
$40 \le t \le 59$	K_t = 8F1BBCDC	$[2^{30} \times \sqrt{5}]$
$60 \le t \le 79$	K_t = CA62C1D6	$[2^{30} \times \sqrt{10}]$

Each f_t, 0<=t<=79, operates on three 32-bit words B, C, D and produces a 32-bit word as output. f_t(B,C,D) is defined as follows: for words B, C, D,

Step	Function Name	Function Value
($0 \le t \le 19$)	$f_1 = f(t,B,C,D)$	$(B \wedge C) \vee (B' \wedge D)$
($20 \le t \le 39$)	$f_2 = f(t,B,C,D)$	$B \oplus C \oplus D$
($40 \le t \le 59$)	$f_3 = f(t,B,C,D)$	$(B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$
($60 \le t \le 79$)	$f_4 = f(t,B,C,D)$	$B \oplus C \oplus D$

Instead of just splitting the input block into 32-bit words and using them directly, SHA-1 shuffles and mixes them using rotates & XOR's to form a more complex input, and greatly increases the difficulty of finding collisions. The procedure is as follows:

- Consider each message block as a chunk of 512 bits and break it into sixteen 32-bit big endian words: W[i], 0 ≤ i ≤ 15
- Each chunk will be put through a little function that will create 80 words from the 16 current ones, which is like a loop.

for i **from** 16 to 79

w[i]:= (w[i-3] xor w[i-8] xor w[i-14] xor w[i-16]) leftrotate 1

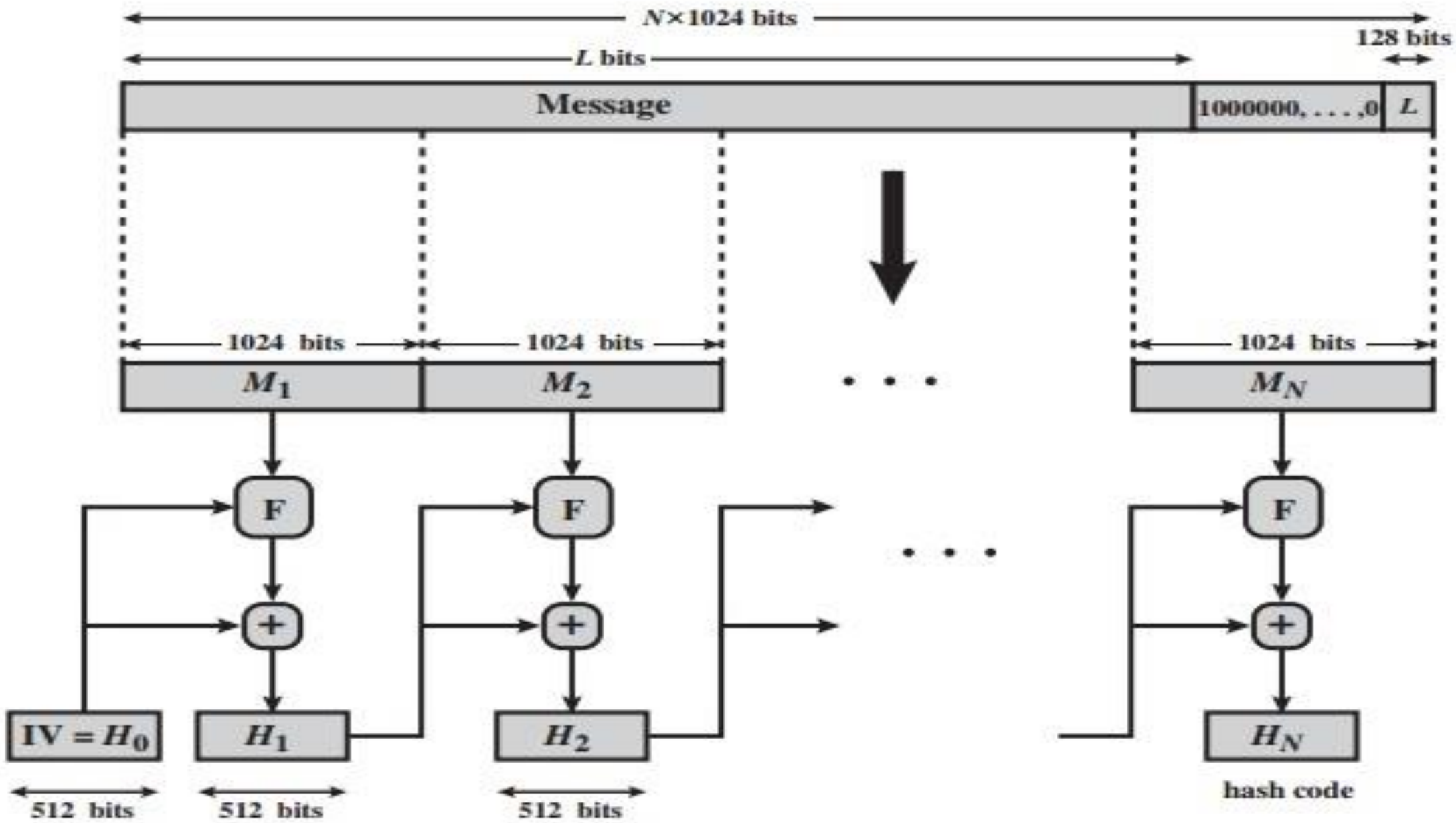
That means for the first time through the loop we want the words numbered= 13, 8, 2 and 0.
The next time through the loop we want words= 14, 9, 13 and 1.

Revised Secure Hash Standard

- In 2002, NIST produced a revised version of the standard, FIPS 180-2, that defined three new versions of SHA, with hash value lengths of 256, 384, and 512 bits, known as SHA-256, SHA-384, and SHA-512, respectively.
- Collectively, these hash algorithms are known as **SHA-2**. These new versions have the same underlying structure and use the same types of modular arithmetic and logical binary operations as SHA-1.
- A revised document was issued as FIP PUB 180-3 in 2008, which added a 224-bit version.
- The structure & detail is similar to SHA-1, hence analysis should be similar, but security levels are rather higher.
- Since 2005, SHA-1 has not been considered secure against well-funded opponents, and since 2010 many organizations have recommended its replacement by SHA-2 or SHA-3. Microsoft, Google, Apple and Mozilla have all announced that their respective browsers will stop accepting SHA-1 SSL certificates by 2017.

SHA-512 Logic

- The algorithm takes as input a message with a maximum length of less than 2^{128} bits and produces as output a 512-bit message digest. The input is processed in 1024-bit blocks.
- Step 1: Append padding bits. The message is padded so that its length is congruent to 896 modulo 1024 [length ≡ 896 (mod 1024)]. Padding is always added, even if the message is already of the desired length. Thus, the number of padding bits is in the range of 1 to 1024. The padding consists of a single 1-bit followed by the necessary number of 0-bits.



+ = word-by-word addition mod 2^{64}

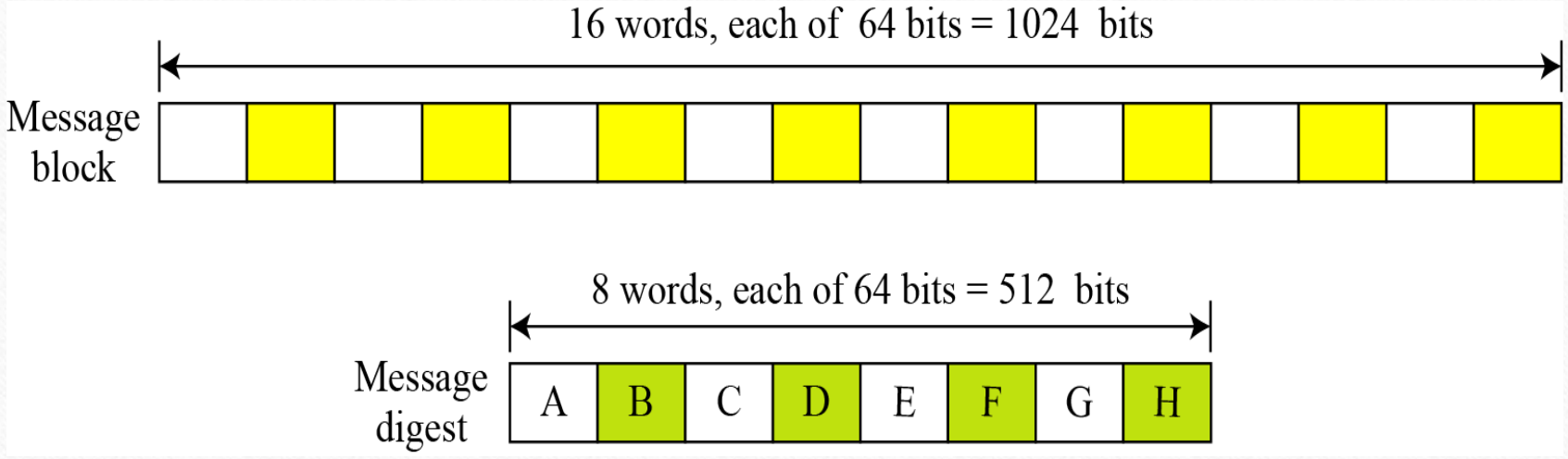
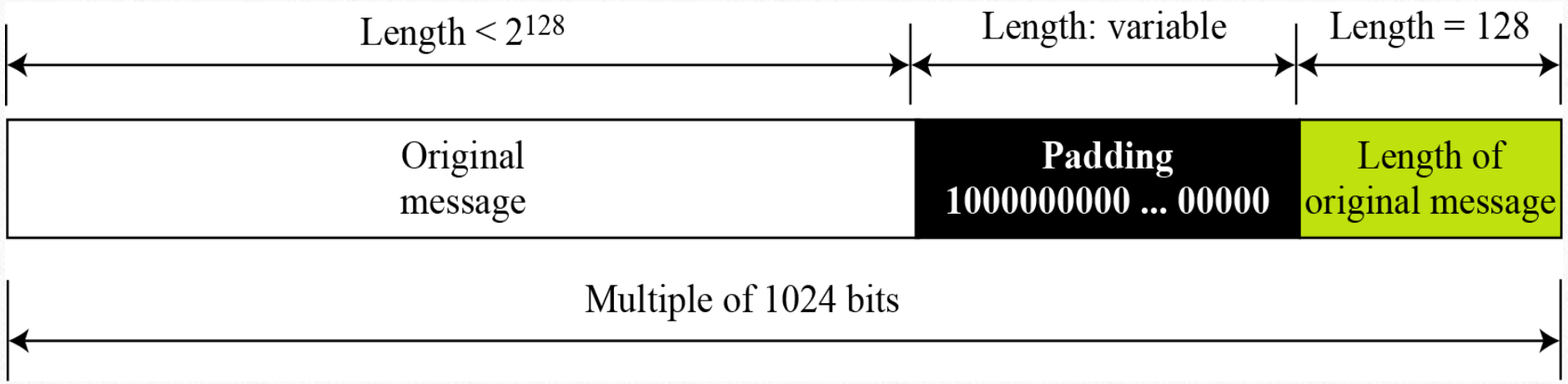


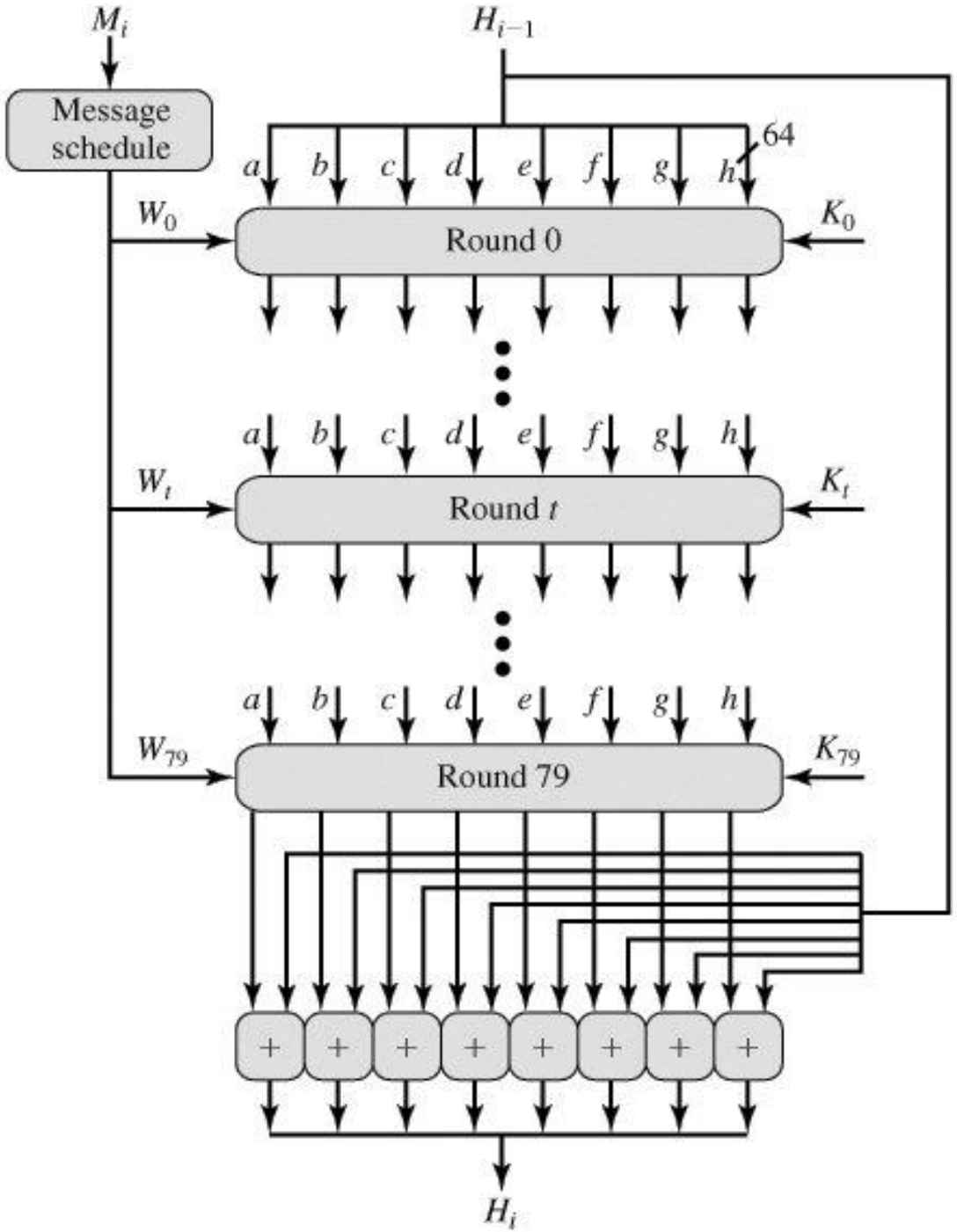
Figure: A message block and the digest as words



- Step 2: Append length. A block of 128 bits is appended to the message. This block is treated as an unsigned 128-bit integer (most significant byte first) and contains the length of the original message (before the padding).

- Step 3: Initialize hash buffer. A 512-bit buffer is used to hold intermediate and final results of the hash function. The buffer can be represented as eight 64-bit registers (a, b, c, d, e, f, g, h). These registers are initialized to the following 64-bit integers (hexadecimal values):

  a = 6A09E667F3BCC908
  b = BB67AE8584CAA73B
  c = 3C6EF372FE94F82B
  c = A54FF53A5F1D36F1
  e = 510E527FADE682D1
  f = 9B05688C2B3E6C1F
  g = 1F83D9ABFB41BD6B
  h = 5BE0CDI9137E2179



- **Step 4:** Process message in 1024-bit (128-word) blocks. The heart of the algorithm is a module that consists of 80 rounds; this module is labeled F.
- The Compression function consists of 80 rounds
  - updating a 512-bit buffer
  - using a 64-bit value Wt derived from the current message block
  - and a round constant based on cube root of first 80 prime numbers
- **Step 5:** Output. After all N 1024-bit blocks have been processed, the output from the Nth stage is the 512-bit message digest.

We can summarize the behavior of SHA-512 as follows:

$$H0 = \text{IV}$$

$$Hi = \text{SUM}_{64}(H_{i-1}, \text{abcdefgh}_i)$$

$$MD = H_N$$

where

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

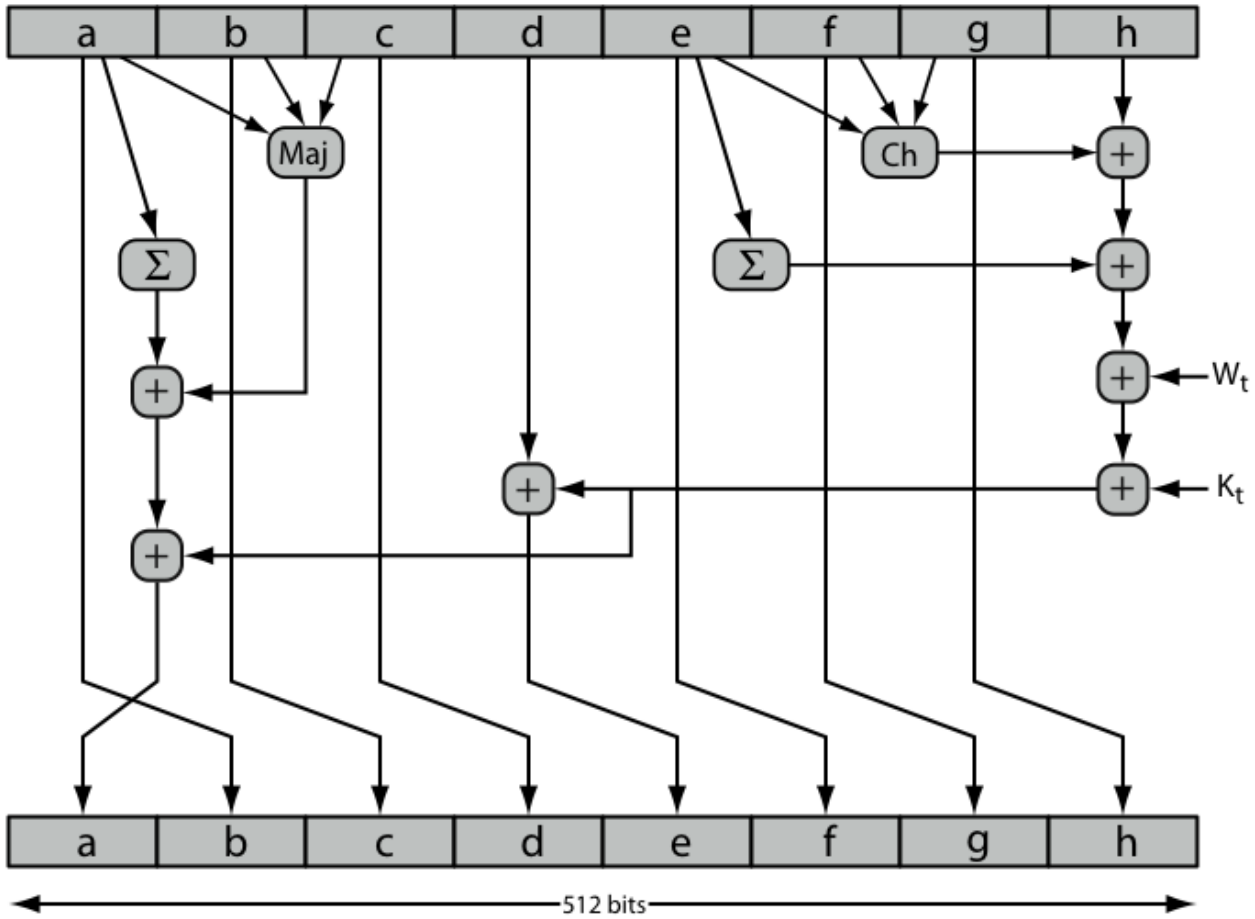
abcdefgh$i$ = the output of the last round of processing of the i^{th} message block

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

SUM_{64} = Addition modulo 2^{64} performed separately on each word of the pair of inputs

MD = final message digest value

SHA-512 Compression Function



Ch(e,f,g) = (e AND f) XOR (NOT e AND g)

Maj(a,b,c) = (a AND b) XOR (a AND c) XOR (b AND c)

∑(a) = ROTR(a,28) XOR ROTR(a,34) XOR ROTR(a,39)

∑(e) = ROTR(e,14) XOR ROTR(e,18) XOR ROTR(e,41)

+ = addition modulo 2^{64}

K_t = a 64-bit additive constant

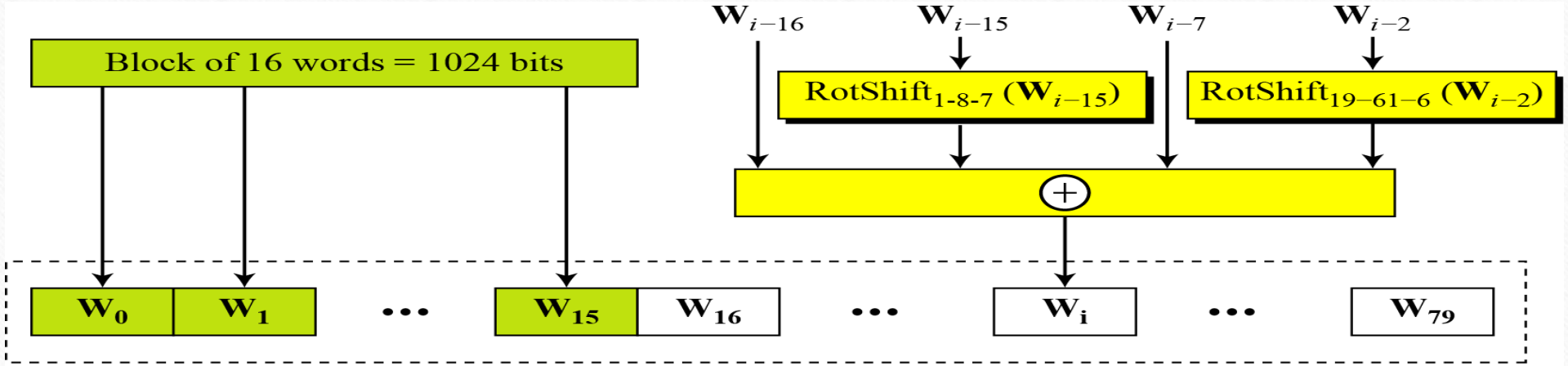
W_t = a 64-bit word derived from the current 512-bit input block.

Each round also makes use of an additive constant Kt where 0 t 79 indicates one of the 80 rounds. These words represent the first sixty-four bits of the fractional parts of the cube roots of the first eighty prime numbers. The constants provide a "randomized" set of 64-bit patterns, which should eliminate any regularities in the input data.

SHA-512 Initial Values & Word Expansion

Buffer	Value (in Hexadecimal)	Buffer	Value (in Hexadecimal)
A_0	6A09E667F3BCC908	E_0	510E527FADE682D1
B_0	BB67AE8584CAA73B	F_0	9B05688C2B3E6C1F
C_0	3C6EF372FE94F82B	G_0	1F83D9ABFB41BD6B
D_0	A54FF53A5F1D36F1	H_0	5BE0CD19137E2179

Table : Values of constants in message digest initialization of 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.

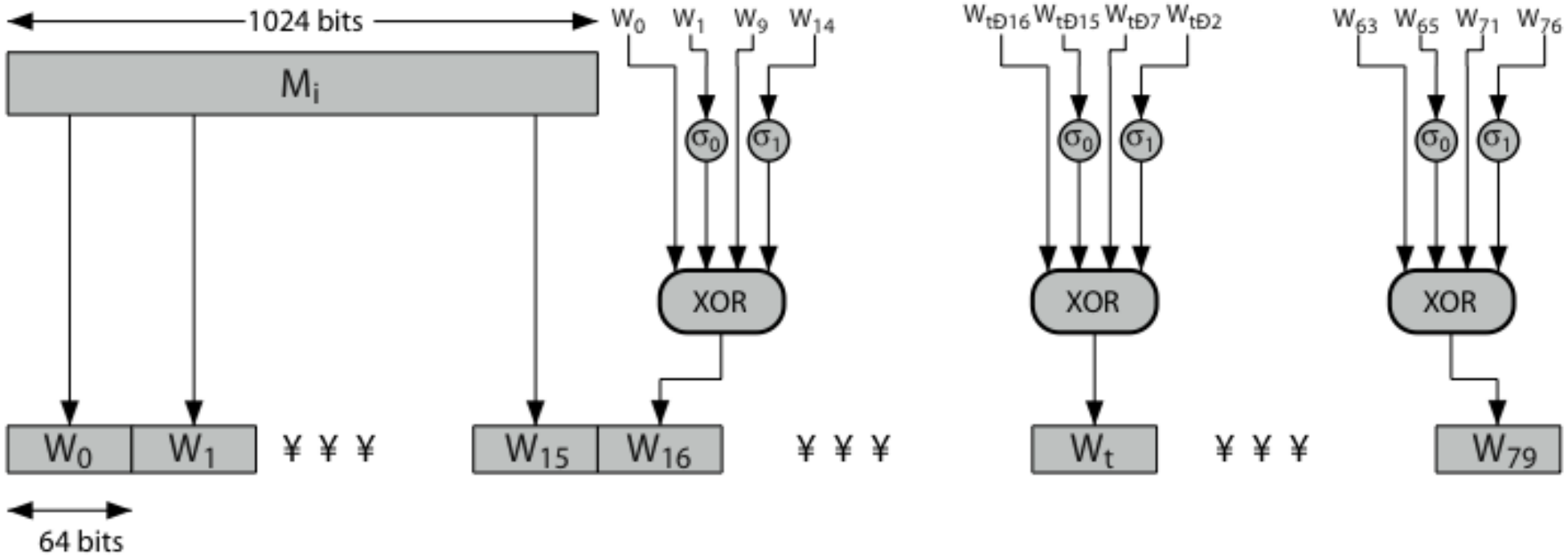
The first 16 values of W_t are taken directly from the 16 words of the current block. The remaining values are defined as follows:

$$W_t = \sigma_1^{512}(W_{t-2}) + W_{t-7} + \sigma_0^{512}(W_{t-15}) + W_{t-16}$$

Where

$$\sigma_0^{512}(x) = \text{ROTR}^1(x) \oplus \text{ROTR}^8(x) \oplus \text{SHR}^7(x)$$
$$\sigma_1^{512}(x) = \text{ROTR}^{19}(x) \oplus \text{ROTR}^{61}(x) \oplus \text{SHR}^6(x)$$

$\text{ROTR}^n(x)$ = circular right shift (rotation) of the 64-bit argument x by n bits
$\text{SHR}^n(x)$ = left shift of the 64-bit argument x by n bits with padding by zeros on the right



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

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

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

The fraction part: $(6C44198C4A475817)_{16}$

Example using SHA-512

ASCII characters: “abc”, which is equivalent to the following 24-bit binary string:

01100001 01100010 01100011 = 616263 in Hexadecimal

The original length is 24 bits, or a hexadecimal value of 18. The 1024-bit message block, in hexadecimal, is

616263800000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000018

W0 = 6162638000000000
W1 = 0000000000000000
W2 = 0000000000000000
W3 = 0000000000000000
W4 = 0000000000000000
*W*10 = 0000000000000000
*W*11 = 0000000000000000
*W*12 = 0000000000000000
W5 = 0000000000000000
W6 = 0000000000000000
W7 = 0000000000000000
W8 = 0000000000000000
W9 = 0000000000000000
*W*13 = 0000000000000000
*W*14 = 0000000000000000
*W*15 = 0000000000000018

March 27, 2023

Example using SHA-512

The following table shows the initial values of these variables and their values after each of the first two rounds.

a	6a09e667f3bcc908	f6afceb8bcfcddf5	1320f8c9fb872cc0
b	bb67ae8584caa73b	6a09e667f3bcc908	f6afceb8bcfcddf5
c	3c6ef372fe94f82b	bb67ae8584caa73b	6a09e667f3bcc908
d	a54ff53a5f1d36f1	3c6ef372fe94f82b	bb67ae8584caa73b
e	510e527fade682d1	58cb02347ab51f91	c3d4ebfd48650ffa
f	9b05688c2b3e6c1f	510e527fade682d1	58cb02347ab51f91
g	1f83d9abfb41bd6b	9b05688c2b3e6c1f	510e527fade682d1
h	5be0cd19137e2179	1f83d9abfb41bd6b	9b05688c2b3e6c1f

The process continues through 80 rounds. The output of the final round is

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

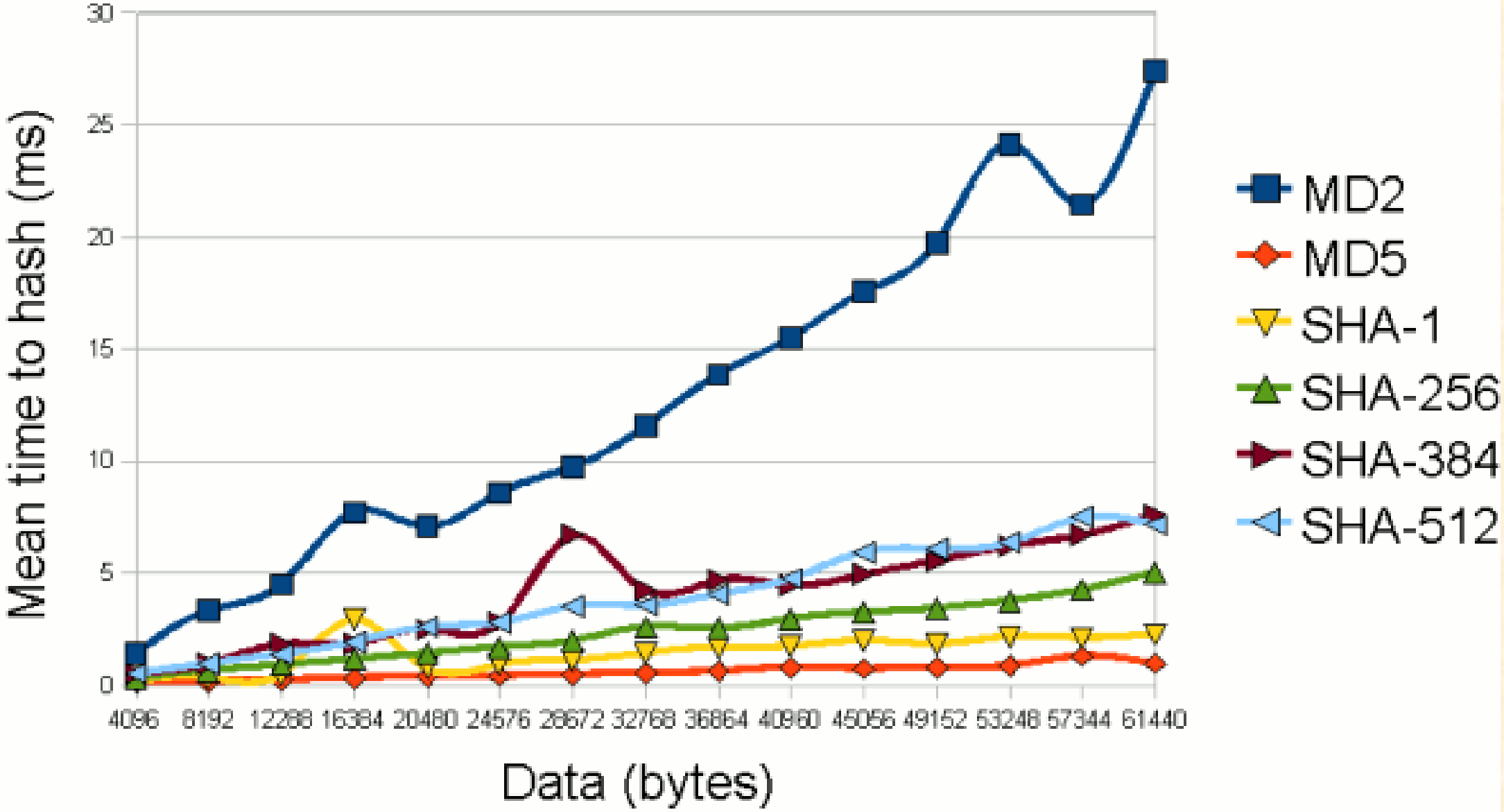
The hash value is then calculated as

H1,7 = 5be0cd19137e2179 + ceb9fc3691ce8326 = 2a9ac94fa54ca49f
H1,6 = 1f83d9abfb41bd6b + 25c96a7768fb2aa3 = 454d4423643ce80e
H1,5 = 9b05688c2b3e6c1f + 9bb4d39778c07f9e = 36ba3c23a3feebbd
*H*1,4 = 510e527fade682d1 + d08446aa79693ed7 = 2192992a274fc1a8
H1,3 = a54ff53a5f1d36f1 + 654ef9abec389ca9 = 0a9eeee64b55d39a
H1,2 = 3c6ef372fe94f82b + d67806db8b148677 = 12e6fa4e89a97ea2
H1,1 = bb67ae8584caa73b + 10d9c4c4295599f6 = cc417349ae204131
H1,0 = 6a09e667f3bcc908 + 73a54f399fa4b1b2 = ddaf35a193617aba

The resulting 512-bit message digest is

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

Speed of secure hash functions



Algorithm and variant		Output size (bits)	Internal state size (bits)	Block size (bits)	Max message size (bits)	Word size (bits)	Rounds	Bitwise operations	Collisions found	Example Performance (MiB/s)[8]
MD5 (as reference)		128	128	512	$2^{64} - 1$	32	64	and,or,xor,rot	Yes	335
SHA-0		160	160	512	$2^{64} - 1$	32	80	and,or,xor,rot	Yes	-
SHA-1		160	160	512	$2^{64} - 1$	32	80	and,or,xor,rot	Theoretical attack (2^{61})[9]	192
SHA-2	*SHA-224* *SHA-256*	224 256	256	512	$2^{64} - 1$	32	64	and,or,xor,shr,rot	None	139
	SHA-384 *SHA-512* *SHA-512/224* *SHA-512/256*	384 512 224 256	512	1024	$2^{128} - 1$	64	80	and,or,xor,shr,rot	None	154
SHA-3		224/256/384/512	1600 (5×5 array of 64-bit words)	1152/1088/832/576		64	24	and,xor,not,rot	None	

HMAC

- In cryptography, a keyed-hash message authentication code (HMAC) is a specific type of message authentication code (MAC) involving a cryptographic hash function and a secret cryptographic key.

- Any cryptographic hash function, such as MD5 or SHA-1, may be used in the calculation of an HMAC; the resulting MAC algorithm is termed HMAC-MD5 or HMAC-SHA1 accordingly. The cryptographic strength of the HMAC depends upon the cryptographic strength of the underlying hash function, the size of its hash output, and on the size and quality of the key.

Objectives of HMAC

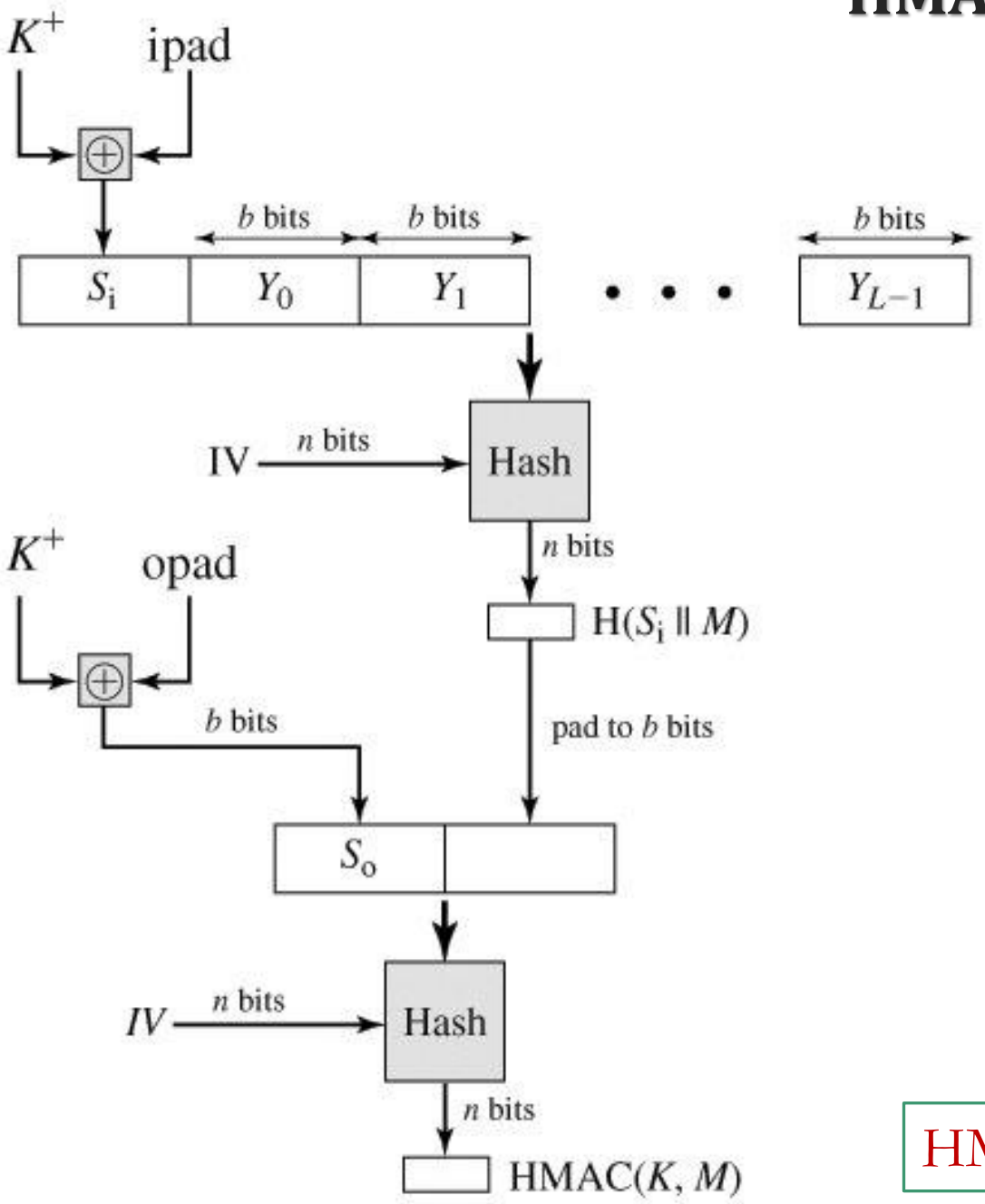
In recent years, there has been increased interest in developing a MAC derived from a cryptographic hash function. The motivations for this interest are

1. Cryptographic hash functions such as MD5 and SHA-1 generally execute faster in software than symmetric block ciphers such as DES.
2. Library code for cryptographic hash functions is widely available.

RFC 2104 lists the following design objectives for HMAC:

- To use, without modifications, available hash functions. In particular, hash functions that perform well in software, and for which code is freely and widely available.
- To allow for easy replaceability of the embedded hash function in case faster or more secure hash functions are found or required.
- To preserve the original performance of the hash function without incurring a significant degradation.
- To use and handle keys in a simple way.
- To have a well understood cryptographic analysis of the strength of the authentication mechanism based on reasonable assumptions about the embedded hash function.

HMAC Algorithm



H = embedded hash function (e.g., MD5, SHA-1, RIPEMD-160)

IV = initial value input to hash function

M = message input to HMAC(including the padding specified in the embedded hash function)

Yi = ith block of M, $0 \leq i \leq (L - 1)$

L = number of blocks in M

b = number of bits in a block

n = length of hash code produced by embedded hash function

K = secret key recommended length is $\geq$ n; if key length is greater than b; the key is input to the hash function to produce an n-bit key

K+ = K padded with zeros on the left so that the result is b bits in length

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

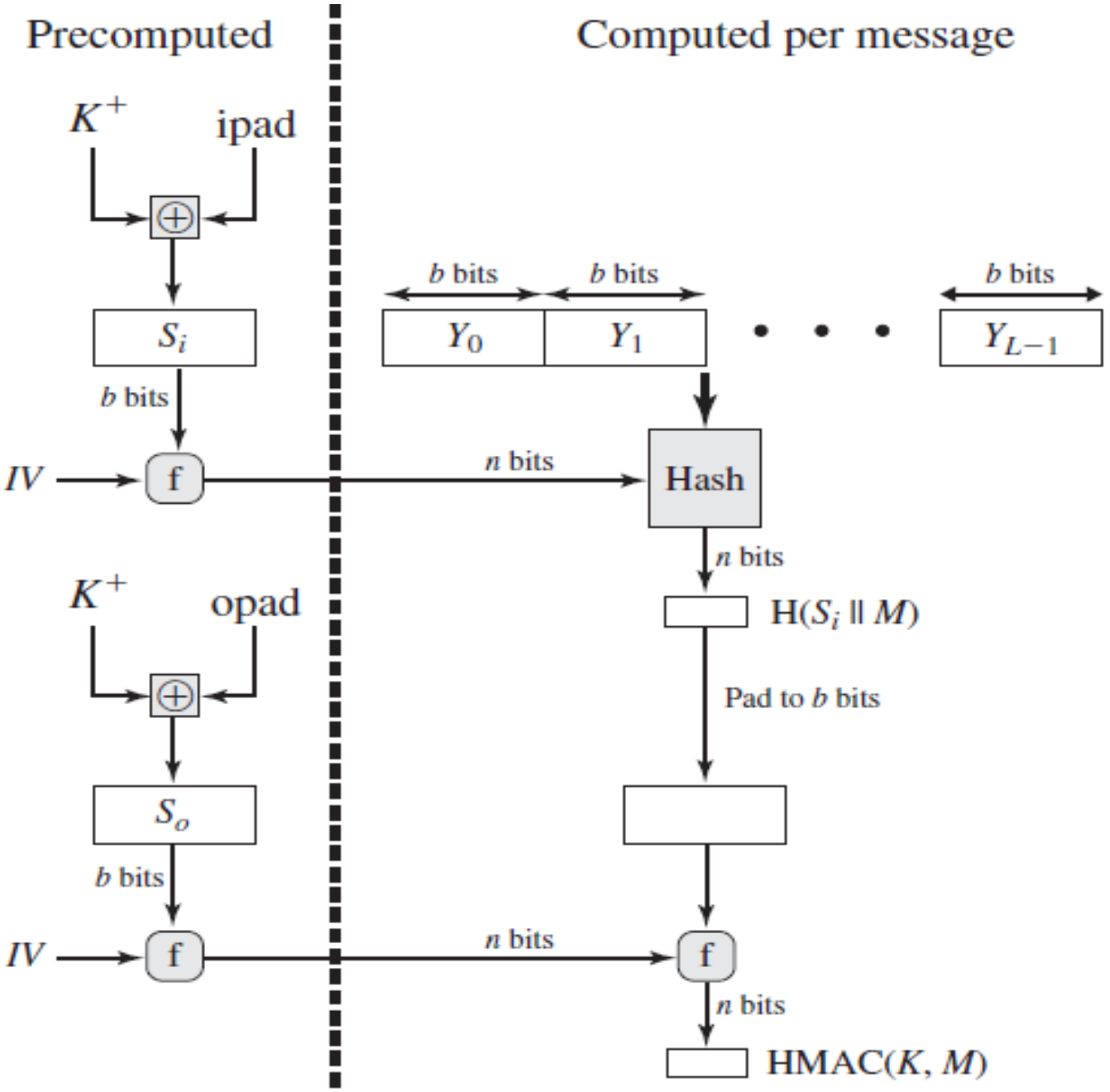
opad = 01011100 (5C in hexadecimal) repeated b/8 times

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

In words, HMAC can be described as:

1. Append zeros to the left end of K to create a b-bit string K+(e.g., if K is of length 160 bits and b = 512 then K will be appended with 44 zero bytes 0 x 00).
2. XOR (bitwise exclusive-OR) K+ with ipad to produce the b-bit block Si.
3. Append M to Si.
4. Apply H to the stream generated in step 3.
5. XOR K+ with opad to produce the b-bit block So
6. Append the hash result from step 4 to So
7. Apply H to the stream generated in step 6 and output the result.

- The XOR with ipad results in flipping one-half of the bits of K. Similarly, the XOR with opad results in flipping one-half of the bits of K, but a different set of bits. In effect, by passing Si and So, through the compression function of the hash algorithm, we have pseudorandomly generated two keys from K.
- HMAC should execute in approximately the same time as the embedded hash function for long messages.



A more efficient implementation is possible, as shown. Two quantities are precomputed. A more efficient implementation is possible, as shown. Two quantities are precomputed.

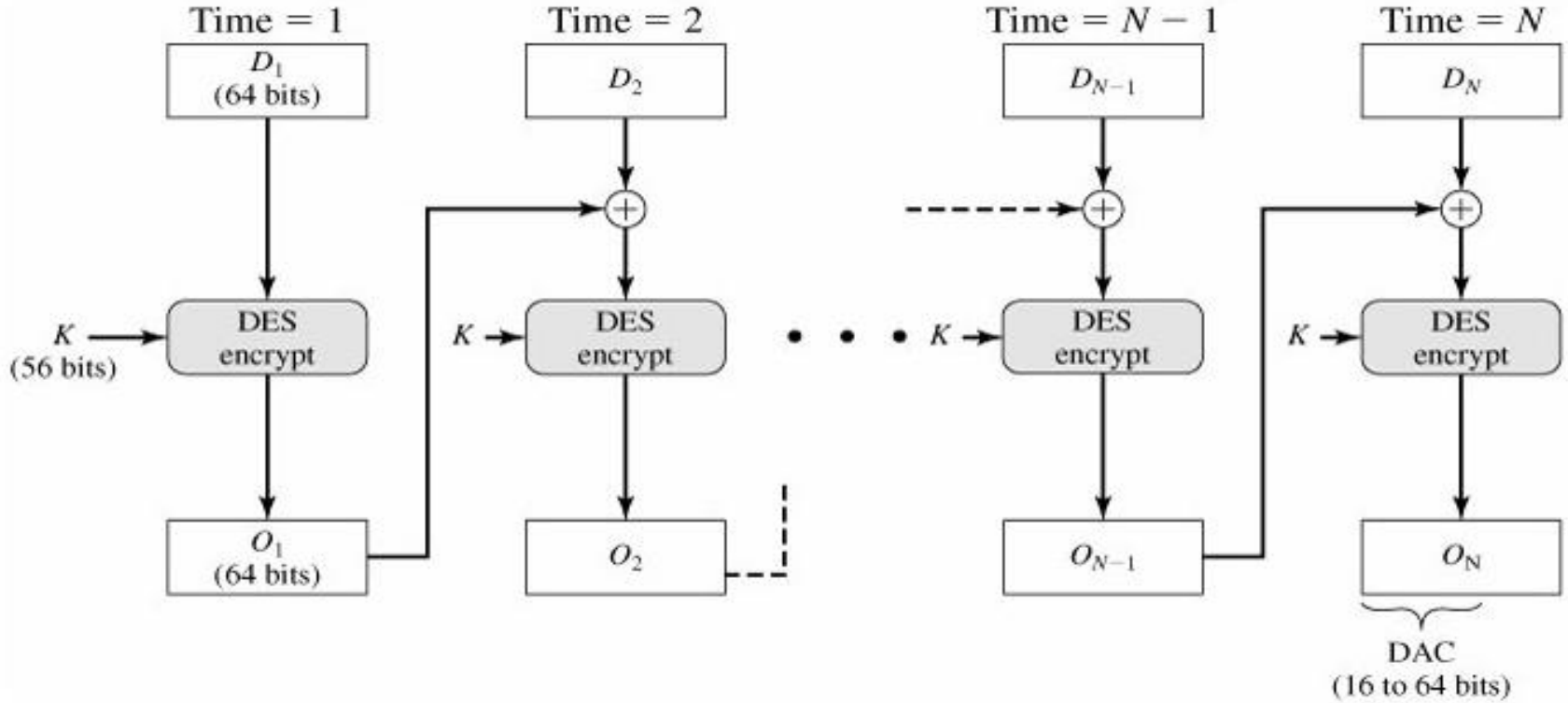
$$f(IV, (K^+ \oplus ipad))$$
$$f(IV, (K^+ \oplus opad))$$

where f(cv, block) is the compression function for the hash function, which takes as arguments a chaining variable of bits and a block of bits and produces a chaining variable of bits.

This efficient implementation is especially worthwhile if most of the messages for which a MAC is computed are short.

Data Authentication Algorithm (DAA)

- The Data Authentication Algorithm, based on DES, has been one of the most widely used MACs for a number of years. The algorithm is both a FIPS publication (FIPS PUB 113) and an ANSI standard (X9.17).
- The algorithm can be defined as using the cipher block chaining (CBC) mode of operation of DES shown below with an initialization vector of zero



The data (e.g., message, record, file, or program) to be authenticated are grouped into contiguous 64-bit blocks: D_1, D_2,..., D_N. If necessary, the final block is padded on the right with zeroes to form a full 64-bit block. Using the DES encryption algorithm, E, and a secret key, K, a data authentication code (DAC) is calculated as follows:

$$O_1 = E(K, D_1)$$

$$O_2 = E(K, [D_2 \oplus O_1])$$

$$O_3 = (K, [D_3 \oplus O_2])$$

•

•

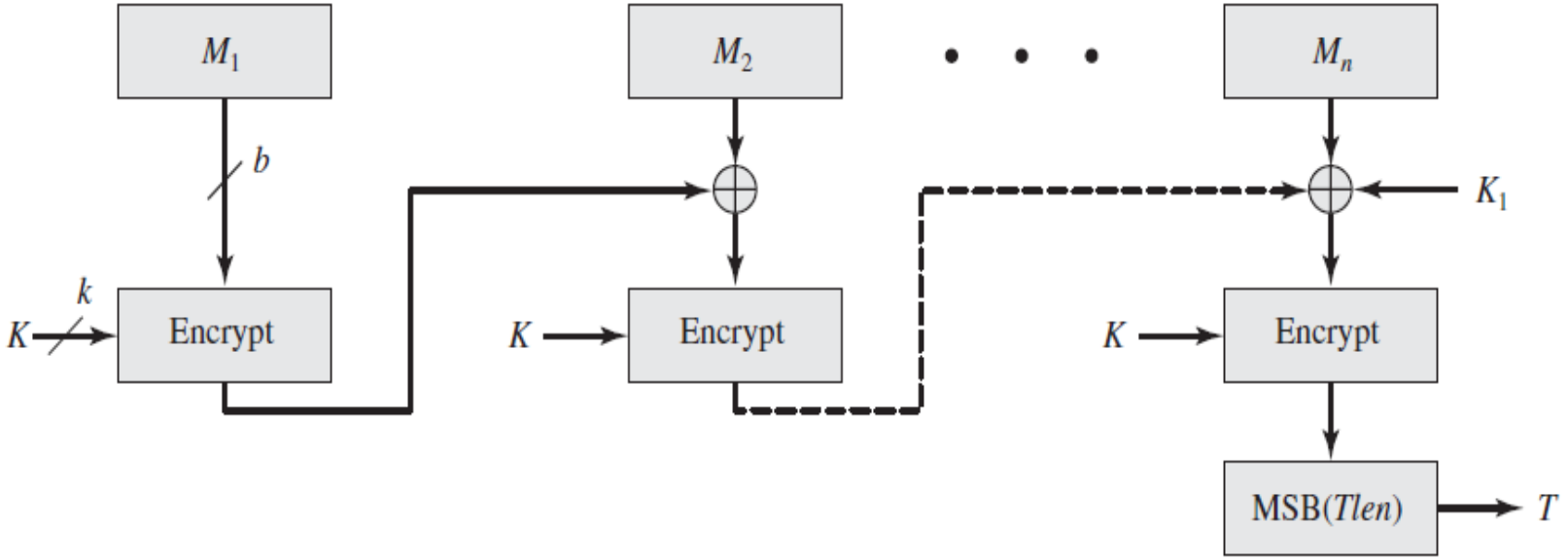
•

$$O_N = E(K, [D_N \oplus O_{N1}])$$

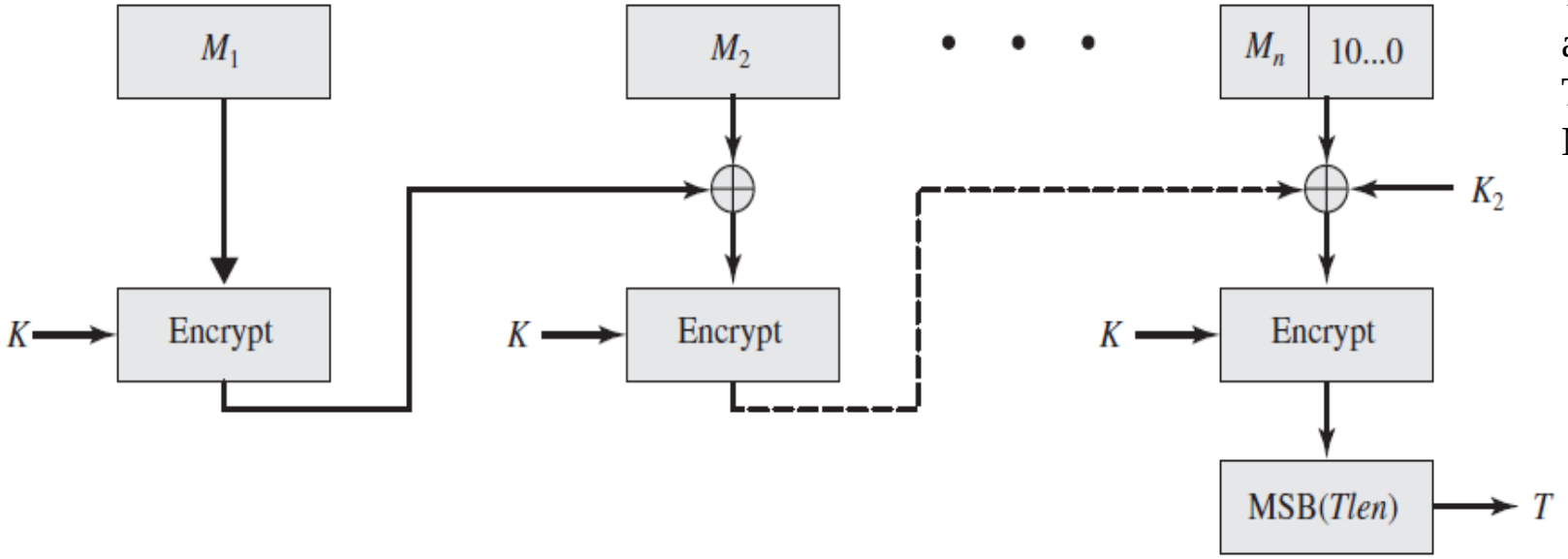
The DAC consists of either the entire block O_N or the leftmost M bits of the block, with $16 \leq M \leq 64$

Cipher-based Message Authentication Code (CMAC)

- In cryptography, CMAC is a block cipher-based message authentication code algorithm. It may be used to provide assurance of the authenticity and, hence, the integrity of binary data. This mode of operation fixes security deficiencies of CBC-MAC (CBC-MAC is secure only for fixed-length messages).
- It is specified in NIST Special Publication 800-38B for use with AES and triple DES.
- when the message is an integer multiple n of the cipher block length b. For AES, b=128, and for triple DES, b=64.
- The message is divided into blocks (M_1, M_2,.....,M_n). The algorithm makes use of a K-bit encryption key and an -bit constant, K_1.
- For AES, the key size is 128, 192, or 256 bits; for triple DES, the key size is 112 or 168 bits



(a) Message length is integer multiple of block size



(b) Message length is not integer multiple of block size

$C_1 = E(K, M1)$
$C_2 = E(K, [M_2 \oplus C_1])$
$C_3 = E(K, [M_3 \oplus C_2])$
.
.
.
$C_n = E(K, [M_n \oplus C_{n-1} \oplus K_1])$
$T = MSB_{Tlen}(C_n)$

T = message authetication code, also referred to as the tag
Tlen = bit length of T
$MSB_s(X)$ = the s leftmost bits of the bit string X