

Unit III: Message Authentication and Hash Functions

(6 Hours) — Cryptography and Network Security

1. Message Authentication

1.1 What is Message Authentication?

In simple language, message authentication answers two questions for the receiver of a message: 'Did this message really come from the person who claims to have sent it?' and 'Has the message been changed on the way?' It does NOT worry about keeping the message secret — that job belongs to encryption. Authentication only worries about trust and integrity: making sure the content is genuine and untouched.

Think of it like a sealed exam answer sheet with the invigilator's signature stamp on it. Anyone can read the answers, but the stamp proves the sheet has not been swapped or altered, and that it really came from the right student.

1.2 Why is it Needed?

Messages travelling over a network can be attacked in several ways even when they are encrypted: an attacker may replay an old captured message, change the order of messages, modify the content in transit, masquerade as someone else, or simply deny later that a message was ever sent (called repudiation). Message authentication mechanisms are designed to defend against exactly these attacks.

Type of Attack	Plain-English Meaning
Masquerade	An attacker pretends to be a legitimate, trusted sender.
Content Modification	Bits of the message are inserted, deleted, or changed in transit.
Sequence Modification	The order of a series of messages is altered or messages are reordered.
Timing Modification	A message is delayed or replayed at a later, inappropriate time (replay attack).
Source Repudiation	Sender later denies having sent the message.
Destination Repudiation	Receiver later denies having received the message.

1.3 Approaches to Message Authentication

There are three broad families of techniques used to authenticate a message:

1. Message Encryption — the ciphertext itself acts as the authenticator; if the message decrypts into something meaningful, it is assumed genuine.

2. Message Authentication Code (MAC) — a small fixed-size 'tag' computed from the message and a shared secret key, sent along with the message.
3. Hash Function — a public algorithm maps the message to a fixed-size digest, which is then protected (e.g. encrypted, or sent over a secure channel) to verify integrity.

Figure: General Model of Message Authentication using a MAC

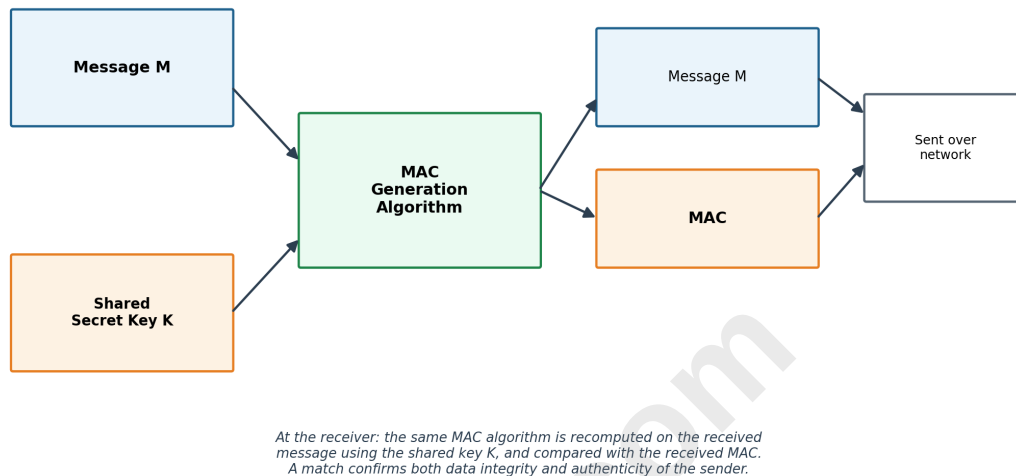


Figure 1.1 — General Model of Message Authentication using a MAC

1.4 Message Authentication Code (MAC)

A MAC is generated by feeding the message M and a shared secret key K into a MAC algorithm: $MAC = C(K, M)$. The sender appends this MAC to the message. The receiver, who also holds the key K, recomputes the MAC on the received message and compares it with the MAC received. If they match, the message is accepted as authentic and unaltered; the matching also proves it came from someone who knows the secret key.

Example 1.1 — Simple MAC computation idea

Suppose the MAC algorithm is defined (for teaching purposes) as: $MAC = (\text{Sum of ASCII values of characters in } M) \bmod K$, where $K = 100$ is the shared secret key. Let the message $M = \text{"HI"}$.

Solution:

ASCII value of 'H' = 72, ASCII value of 'I' = 73.

Sum = 72 + 73 = 145.

MAC = 145 mod 100 = 45.

The sender transmits $M = \text{"HI"}$ along with $\text{MAC} = 45$. The receiver, who also knows $K = 100$, recomputes the same sum and mod operation. If the recomputed value also equals 45, the message is accepted as authentic; any change to even a single character in transit would change the ASCII sum and therefore the MAC, alerting the receiver to tampering.

Note: A MAC is NOT a digital signature. A MAC uses a single shared secret key known to both sender and receiver (symmetric), so it cannot provide non-repudiation — either party could have computed it. Digital signatures (covered in Section 6) solve this using public-key cryptography.

2. Hash Functions

2.1 What is a Hash Function?

In plain words, a hash function is like a fingerprint machine for data. You feed it a message of any length — one word or a whole book — and it always outputs a fixed-size string of bits, called the hash value, message digest, or simply the 'hash'. Even a tiny one-character change in the input produces a completely different output digest.

2.2 Properties Required of a Good Cryptographic Hash Function

Property	Plain-English Meaning
Fixed output size	Same length digest regardless of input message size (e.g. always 128 or 160 bits).
Ease of computation	$H(M)$ must be quick and efficient to compute for any given message M .
Pre-image resistance (one-way)	Given a digest h , it is computationally infeasible to find any M such that $H(M) = h$.
Weak collision resistance	Given M , infeasible to find a different M' with $H(M) = H(M')$.
Strong collision resistance	Infeasible to find ANY pair of distinct messages M and M' such that $H(M) = H(M')$.
Avalanche effect	A 1-bit change in M should change roughly half the bits of the digest.

2.3 General Structure of Hash Functions

Almost every popular hash function (MD5, SHA-1, and their relatives) is built the same way, known as the iterative or Merkle–Damgård structure. The message is padded and split into fixed-size blocks. A compression function f repeatedly combines each block with a running

'chaining value', starting from a fixed Initial Value (IV). The chaining value after the last block becomes the final digest.

Figure: Iterative (Merkle-Damgard) Structure of a Cryptographic Hash Function

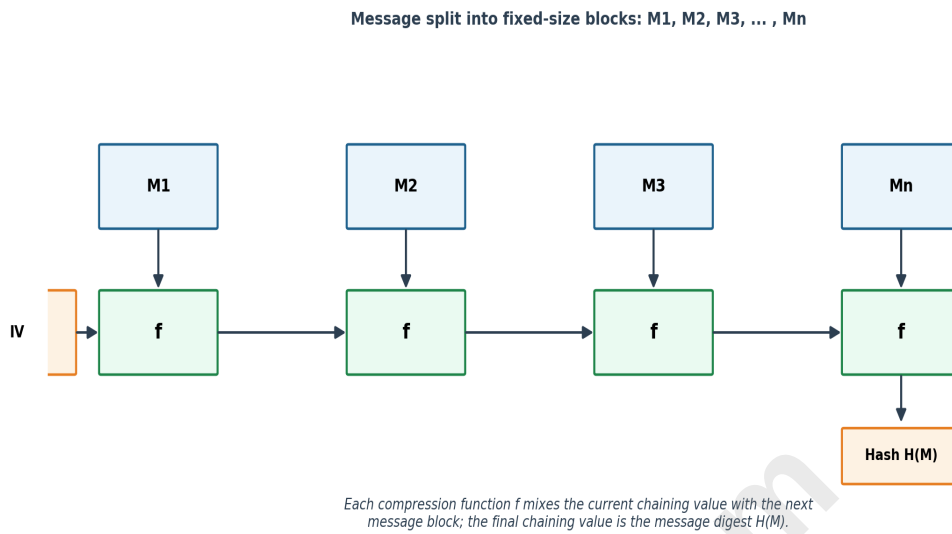


Figure 2.1 — Iterative (Merkle-Damgard) Structure of a Cryptographic Hash Function

2.4 Simple Hash Function Example (illustrative, not cryptographically secure)

Example 2.1 — A toy XOR-based hash

A very simple (insecure, for teaching only) hash function splits the message into 8-bit blocks and XORs them together to produce an 8-bit digest. Let message $M = "AB"$ where $A = 01000001$ and $B = 01000010$ in binary (ASCII).

Solution:

Block 1 = 01000001

Block 2 = 01000010

XOR them bitwise: $01000001 \text{ XOR } 01000010 = 00000011$

Digest $H(M) = 00000011$ (decimal 3).

This illustrates the basic idea of combining blocks into a fixed-size output, but a real cryptographic hash uses far more complex non-linear mixing (rounds of bitwise functions, modular addition, and shifting) so that it is infeasible to reverse or find collisions, unlike this simple XOR example.

Note: Simple XOR-based 'hashes' like the one above are easy to break by an attacker because many different messages XOR to the same result. Real-world hash functions such as MD5 and SHA-1 use many rounds of

non-linear bit operations specifically to prevent this kind of weakness.

3. Message Digests: MD4 and MD5

3.1 MD4

MD4 (Message Digest 4), designed by Ron Rivest in 1990, was one of the earliest widely used hash functions. It produces a 128-bit digest from a message of any length, processed in 512-bit blocks using three rounds of 16 steps each. MD4 is now considered cryptographically broken (collisions can be found quickly) and is not used in modern systems, but it is the direct ancestor of MD5 and SHA, so understanding it helps in understanding the later algorithms.

3.2 MD5 — Step by Step

MD5 (Message Digest 5), also by Rivest (1992), strengthened MD4 by adding a fourth round and other refinements. It still produces a 128-bit (16-byte) digest, usually shown as 32 hexadecimal characters. The algorithm proceeds in five simple-to-understand steps:

1. **Padding:** Bits are appended to the message (a single 1-bit followed by 0-bits) until its length is 64 bits short of being a multiple of 512 bits.
2. **Append length:** A 64-bit field representing the length of the original message (before padding) is appended, making the total length an exact multiple of 512 bits.
3. **Initialize MD buffer:** Four 32-bit registers A, B, C, D are set to fixed standard initial hexadecimal values.
4. **Process message in 512-bit blocks:** Each block goes through 4 rounds of 16 steps (64 steps total), each round using a different non-linear function (F, G, H, I) along with modular addition and left-rotation, updating the buffer.
5. **Output:** After the final block, the contents of buffer A, B, C, D (128 bits total) form the message digest.

Figure: MD5 Message Digest Generation (128-bit output, 4 rounds per block)

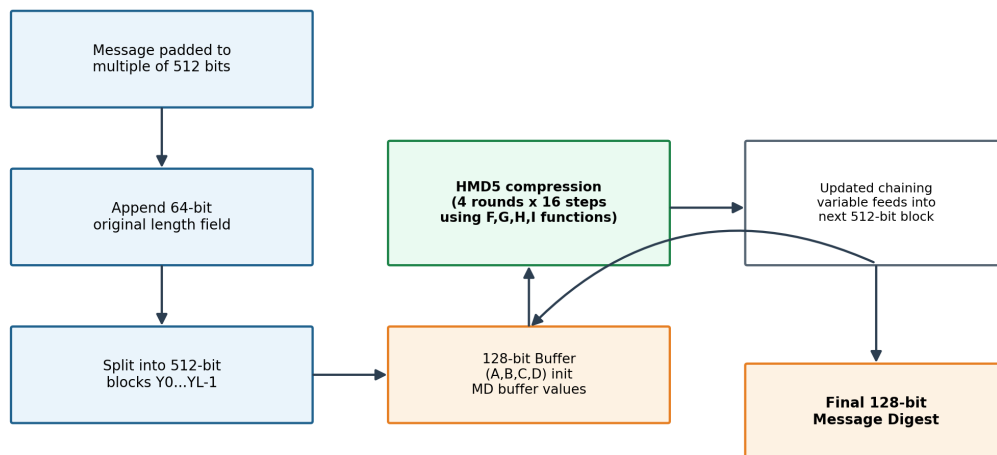


Figure 3.1 — MD5 Message Digest Generation (128-bit output)

Example 3.1 — Conceptual MD5 padding calculation

Suppose a message is 1000 bits long. How many padding bits must be added before the 64-bit length field, so that the total becomes a multiple of 512 bits?

Solution:

We need: $(1000 + \text{padding} + 64) \bmod 512 = 0$

$1000 \bmod 512 = 488$, so currently we are 488 bits into a 512-bit block.

Space remaining in this block before the 64-bit length field = $512 - 488 - 64 = -40$, which is negative, so we must move into the NEXT block.

Total target length before length field = smallest multiple of 512 that is at least $1000 + 1$ (for the mandatory 1-bit) + 64 = 1065, which rounds up to 1536.

Padding bits required = $1536 - 64 - 1000 = 472$ bits (this includes the leading single 1-bit followed by 0-bits).

So 472 padding bits are added, followed by the 64-bit length field, giving a final padded message of exactly 1536 bits = 3 blocks of 512 bits.

Note: MD5 is now considered broken for security-critical use (practical collision attacks exist), so it should not be used for digital signatures or certificates today. It is still occasionally used for simple, non-security checksums such as verifying accidental file corruption during download.

4. Secure Hash Algorithms: SHA-1

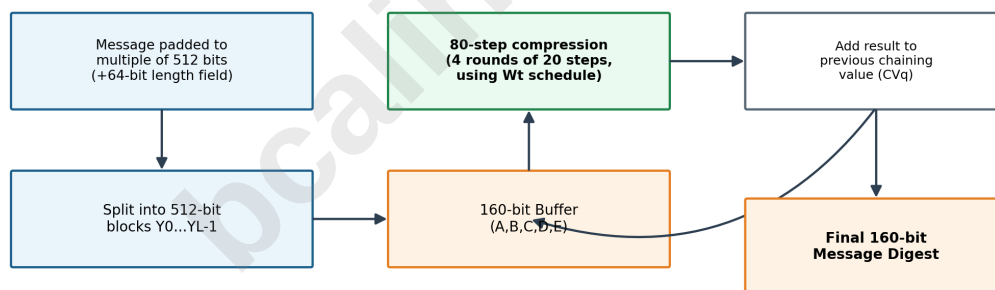
4.1 Overview

SHA-1 (Secure Hash Algorithm 1) was developed by NIST and the NSA, closely modelled on MD4/MD5 but producing a longer 160-bit (20-byte) digest instead of 128 bits. A longer digest means a much larger number of possible outputs, making brute-force collision search significantly harder than for MD5.

4.2 SHA-1 Algorithm Steps

1. Padding and length append: exactly as in MD5, the message is padded to a multiple of 512 bits with a 64-bit length field appended.
2. Initialize buffer: five 32-bit registers A, B, C, D, E are initialized to fixed standard hexadecimal constants (one more register than MD5's four).
3. Message schedule: each 512-bit block is expanded into eighty 32-bit words $W_0 \dots W_{79}$ using an expansion formula.
4. Compression (80 steps): four rounds of 20 steps each update the buffer using round-specific logical functions and constants, plus left-rotations and modular addition.
5. Output: the final A, B, C, D, E values (160 bits total) form the message digest.

Figure: SHA-1 Message Digest Generation (160-bit output, 80 steps per block)



SHA-1 produces a longer 160-bit digest than MD5's 128 bits, giving stronger resistance to brute-force and collision attacks (though SHA-1 is now deprecated).

Figure 4.1 — SHA-1 Message Digest Generation (160-bit output)

4.3 Comparing MD5 and SHA-1

Feature	MD5	SHA-1
Digest length	128 bits	160 bits
Basic processing unit	512-bit blocks	512-bit blocks

Feature	MD5	SHA-1
Number of steps per block	64 (4 rounds x 16)	80 (4 rounds x 20)
Number of buffer registers	4 (A,B,C,D)	5 (A,B,C,D,E)
Resistance to brute force	Weaker (2^{64} effort)	Stronger (2^{80} effort)
Current security status	Broken, not recommended	Deprecated, being phased out for SHA-2/3

Example 4.1 — Why a longer digest matters (brute-force comparison)

Compare, in plain terms, the effort needed for a brute-force collision search (birthday attack) between MD5 (128-bit digest) and SHA-1 (160-bit digest).

Solution:

For an n -bit hash, a birthday-attack style collision search needs roughly $2^{(n/2)}$ attempts on average.

For MD5: $n = 128$, so effort is approximately 2^{64} attempts.

For SHA-1: $n = 160$, so effort is approximately 2^{80} attempts.

2^{80} is roughly $2^{16} = 65,536$ times larger than 2^{64} , meaning SHA-1's birthday-bound brute-force resistance is tens of thousands of times stronger than MD5's, purely due to the longer digest size.

5. HMAC (Hash-based Message Authentication Code)

5.1 Why HMAC?

Plain hash functions like MD5/SHA-1 are public algorithms with no secret key, so by themselves they cannot authenticate a sender — anyone could recompute the same hash. HMAC solves this by combining a hash function with a secret key in a specific, carefully designed way, turning an ordinary hash function into a proper MAC.

5.2 How HMAC Works

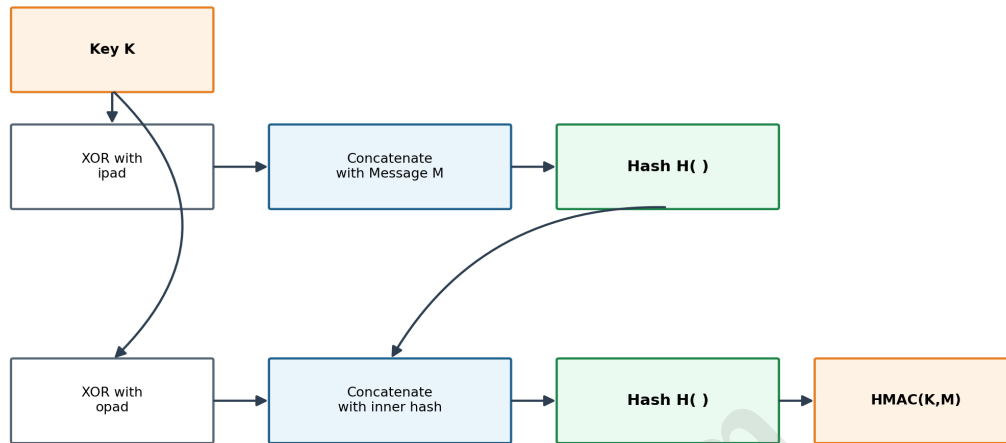
HMAC uses two fixed padding constants: *ipad* (inner pad) and *opad* (outer pad), and applies the hash function H twice, nested inside one another:

$$\text{HMAC}(K, M) = H((K \text{ XOR } \text{opad}) \parallel H((K \text{ XOR } \text{ipad}) \parallel M))$$

In plain words: the key is first XOR-ed with *ipad* and combined with the message, then hashed once (the 'inner hash'). The key is separately XOR-ed with *opad*, combined with the inner hash

result, and hashed again (the 'outer hash'). This double-hashing with two differently-padded copies of the key is what gives HMAC its strong security guarantees, protecting against attacks that work on a single plain hash.

Figure: HMAC (Hash-based Message Authentication Code) Construction



HMAC applies the hash function twice with padded/XOR-ed keys (ipad, opad), giving strong resistance to length-extension and forgery attacks.

Figure 5.1 — HMAC Construction using Inner and Outer Hashing

Example 5.1 — Conceptual HMAC walkthrough

Using a (simplified, illustrative) hash function $H(x) = \text{sum of ASCII codes of } x \text{ mod } 256$, and key $K = "12"$ (ASCII 49, 50), message $M = "OK"$ (ASCII 79, 75). For simplicity, take $\text{ipad} = 1$ and $\text{opad} = 2$ (single numbers added to each key byte, instead of full 64-byte XOR pads, purely to keep the arithmetic easy to follow by hand).

Solution:

Step 1 (inner key): K XOR-like with $\text{ipad} \Rightarrow$ treat as $(49+1, 50+1) = (50, 51)$.

Step 2 (inner hash input): inner key bytes + message bytes = $50 + 51 + 79 + 75 = 255$.

Inner hash $H1 = 255 \text{ mod } 256 = 255$.

Step 3 (outer key): K with $\text{opad} \Rightarrow (49+2, 50+2) = (51, 52)$.

Step 4 (outer hash input): outer key bytes + inner hash = $51 + 52 + 255 = 358$.

HMAC value = $358 \text{ mod } 256 = 102$.

Even though this is a toy example with simplified padding, it captures HMAC's essential idea: the secret key is mixed in twice, once on the inside and once on the outside of the hashing process, so an attacker who does not know K cannot reproduce the correct HMAC value even if they know the hash function and the message.

Note: HMAC can be built using any underlying hash function, giving names like HMAC-MD5, HMAC-SHA1, or HMAC-SHA256. It is widely used today to secure web traffic (TLS), VPNs (IPsec), and API authentication tokens.

6. Digital Signatures

6.1 What is a Digital Signature?

A digital signature is the electronic equivalent of a handwritten signature, but cryptographically much stronger. Unlike a MAC (which uses one shared secret key), a digital signature uses public-key (asymmetric) cryptography: the sender signs with their own private key, and anyone can verify the signature using the sender's public key. This extra property — that only the sender could have created the signature — gives non-repudiation, something a simple MAC cannot provide.

6.2 Why Not Just Sign the Whole Message?

Public-key operations (like RSA) are computationally slow on large amounts of data. So instead of encrypting the entire message with the private key, the sender first computes a fast hash digest of the message, and only encrypts (signs) that small fixed-size digest. This is efficient and still secures the whole message, because any change to the message would change its hash digest.

6.3 Step-by-Step: Signing and Verifying

Signing (sender):

1. Compute the hash digest $H(M)$ of the message using a hash function (e.g. SHA-1).
2. Encrypt the digest using the sender's private key — this encrypted digest is the digital signature.
3. Send the original message M together with the digital signature.

Verifying (receiver):

1. Compute the hash digest $H(M)$ of the received message using the same hash function.
2. Decrypt the received signature using the sender's public key to recover the original digest.
3. Compare the two digest values: if they match, the message is authentic, unaltered, and the sender cannot later deny having sent it.

Figure: Digital Signature Generation and Verification Process

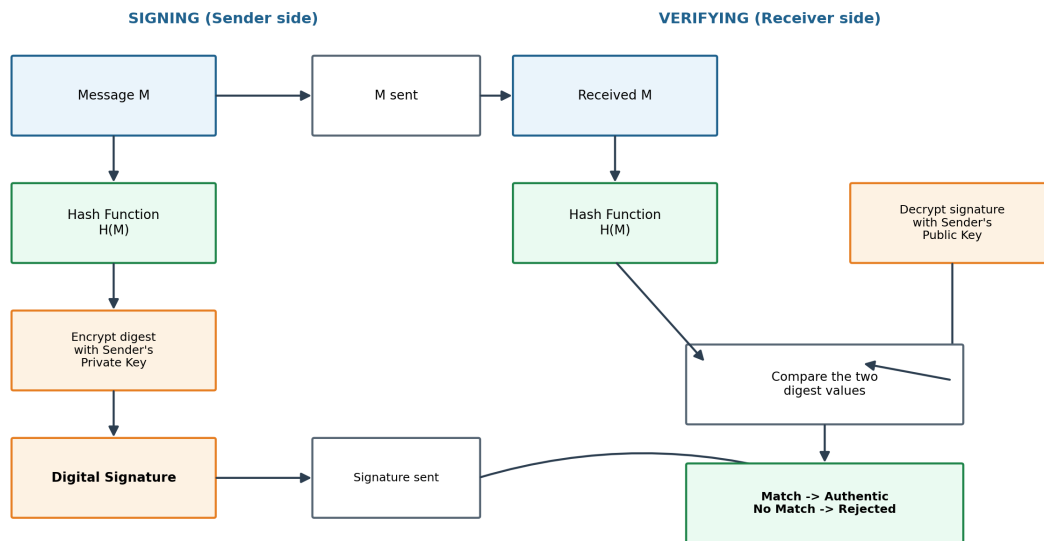


Figure 6.1 — Digital Signature Generation and Verification Process

Example 6.1 — Conceptual digital signature check (small numbers, RSA-style idea)

Using small illustrative RSA-style keys (not secure in real life, only for understanding the arithmetic), let public key $(e, n) = (3, 33)$ and private key $(d, n) = (7, 33)$. Suppose the hash digest of a message works out to the small number $H(M) = 4$. Show how the sender signs it and how the receiver verifies it.

Solution:

Signing (sender uses private key $d=7, n=33$): Signature $S = H(M)^d \bmod n = 4^7 \bmod 33$.

$4^2 = 16$; $4^4 = 16^2 = 256 \bmod 33 = 25$; $4^7 = 4^4 * 4^2 * 4^1 = 25 * 16 * 4 \bmod 33$.

$25 * 16 = 400$, $400 \bmod 33 = 4$ (since $33 * 12 = 396$, remainder 4); then $4 * 4 = 16$.

So Signature $S = 16$.

Verification (receiver uses public key $e=3, n=33$): Recovered digest $= S^e \bmod n = 16^3 \bmod 33$.

$16^2 = 256 \bmod 33 = 25$; $16^3 = 25 * 16 \bmod 33 = 400 \bmod 33 = 4$.

Recovered digest $= 4$, which exactly matches the original $H(M) = 4$, so the signature verifies successfully and the message is accepted as authentic.

Note: Common real-world digital signature algorithms include RSA-based signatures, DSA (Digital Signature Algorithm), and ECDSA (Elliptic Curve DSA). All follow the same hash-then-sign principle shown above, just with much larger, cryptographically secure key sizes in practice.

6.4 MAC vs Digital Signature — Quick Comparison

Aspect	MAC	Digital Signature
Key type	Symmetric (one shared secret key)	Asymmetric (private key to sign, public key to verify)
Non-repudiation	Not provided	Provided
Speed	Generally faster	Generally slower (public-key math)
Typical use	Fast integrity checks between two trusted parties	Legal/contractual signing, certificates, software signing

Key Takeaways

- Message authentication confirms that a message is genuine and unaltered, separate from confidentiality (encryption).
- A MAC uses a shared secret key to produce a small authenticator tag; it cannot prove non-repudiation.
- A cryptographic hash function maps any-length input to a fixed-size digest using an iterative compression structure.
- MD5 produces a 128-bit digest in 4 rounds of 16 steps; it is now considered broken for security use.
- SHA-1 produces a stronger 160-bit digest in 4 rounds of 20 steps (80 total), but is now deprecated in favour of SHA-2/3.
- HMAC combines a hash function with a secret key using nested inner/outer hashing, turning a public hash into a secure MAC.
- Digital signatures use public-key cryptography to sign a message's hash digest, providing authenticity, integrity, AND non-repudiation, unlike a plain MAC.

— End of Unit III —