**UNIT – 3**

**Public Key Cryptography and RSA**

**Short Notes: Evolution of Public-Key Cryptography**

* **Problems in Symmetric Encryption**:
  1. **Key Distribution**: Needs either pre-shared keys or a Key Distribution Center (KDC), which risks compromising secrecy.
  2. **Digital Signatures**: Needed to verify the sender's identity in electronic communication.
* **Diffie and Hellman's Contribution (1976)**:
  1. Proposed a **new cryptographic method** solving both problems.
  2. Introduced the concept of **public-key cryptography(Public key cryptography is a cryptographic system that uses two different but related keys — a public key and a private key — for secure communication.)**, allowing secure communication without shared secrets.
* **Significance**:
  1. Eliminated dependence on KDC.
  2. Enabled secure **authentication and digital signatures**.

**Public-Key Cryptosystems**

Asymmetric algorithms rely on one key for encryption and a different but related key for decryption. These algorithms have the following important characteristic.

• It is computationally infeasible to determine the decryption key given only knowledge of the cryptographic algorithm and the encryption key.

In addition, some algorithms, such as RSA, also exhibit the following characteristic.

• Either of the two related keys can be used for encryption, with the other used for decryption.

A public-key encryption scheme has six ingredients (Figure 9.1a; compare with Figure 2.1).

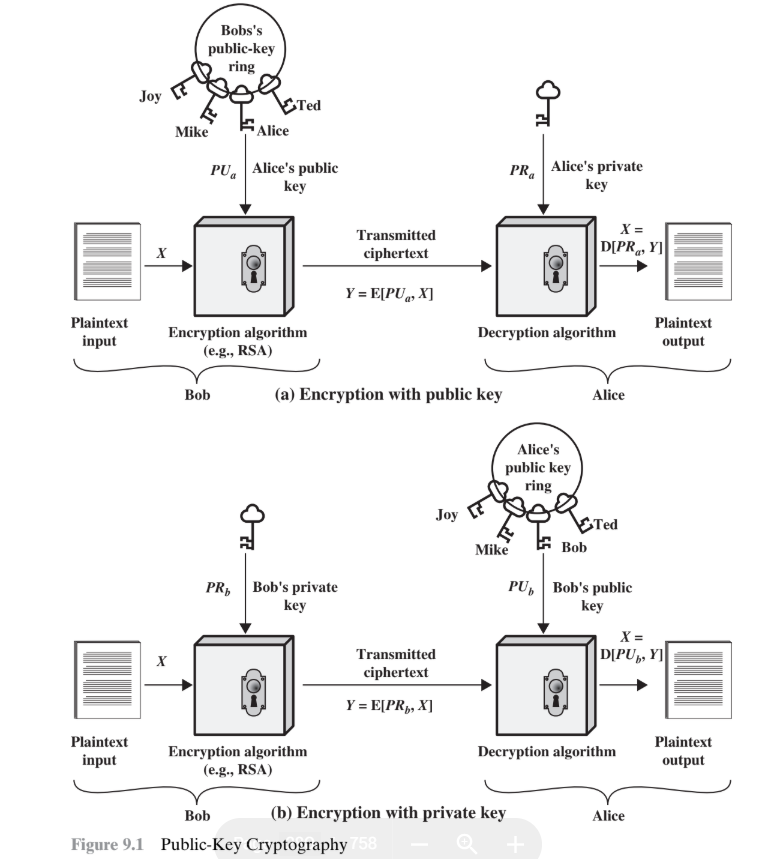
• Plaintext: This is the readable message or data that is fed into the algorithm as input.

Encryption algorithm: The encryption algorithm performs various transformations on the plaintext.

• Public and private keys: This is a pair of keys that have been selected so that if one is used for encryption, the other is used for decryption. The exact transformations performed by the algorithm depend on the public or private key that is provided as input.

• Ciphertext: This is the scrambled message produced as output. It depends on the plaintext and the key. For a given message, two different keys will produce two different ciphertexts.

• Decryption algorithm: This algorithm accepts the ciphertext and the matching key and produces the original plaintext.



The essential steps are the following.

1. Each user generates a pair of keys to be used for the encryption and decryption of messages.

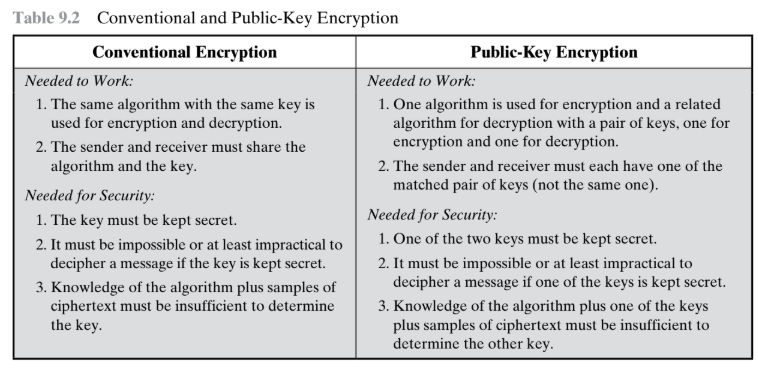
2. Each user places one of the two keys in a public register or other accessible file. This is the public key. The companion key is kept private. As Figure 9.1a suggests, each user maintains a collection of public keys obtained from others.

3. If Bob wishes to send a confidential message to Alice, Bob encrypts the message using Alice’s public key.

4. When Alice receives the message, she decrypts it using her private key. Noother recipient can decrypt the message because only Alice knows Alice’s private key.

To discriminate between the two, we refer to the key used in symmetric encryption as a secret key. The two keys used for asymmetric encryption are referred to as the public key and the private key.

Invariably, the private key is kept secret, but it is referred to as a private key rather than a secret key to avoid confusion with symmetric encryption.



Applications for Public-Key Cryptosystems:

we can classify the use of public-key cryptosystems into three categories

 **Encryption/Decryption**

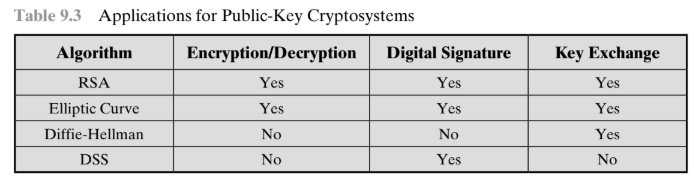
* Sender encrypts message using the **recipient’s public key**.

 **Digital Signature**

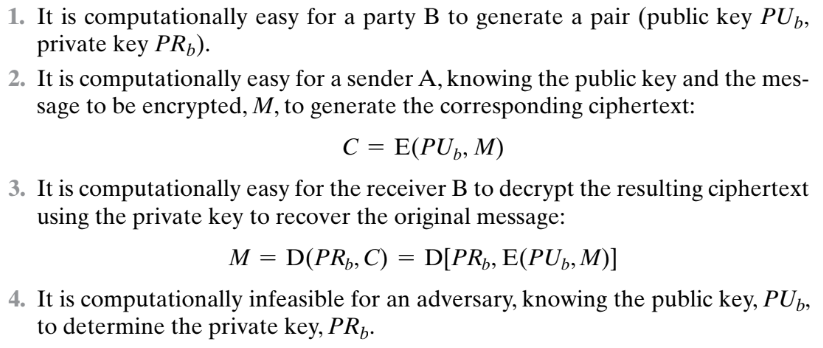
* Sender signs message using their **own private key**.
* Signature is generated using a cryptographic algorithm on the message or its hash.

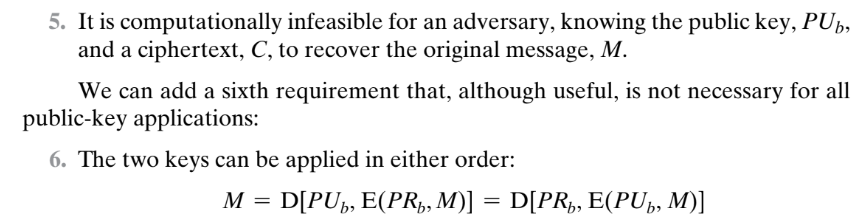
 **Key Exchange**

* Both parties collaborate to exchange a **session key**.
* May involve private keys of one or both parties.

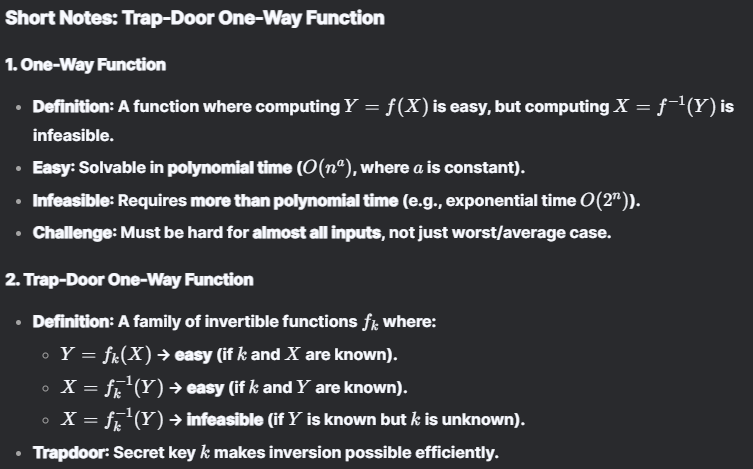


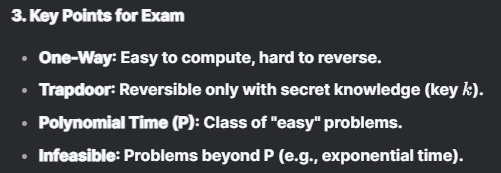
Requirements for Public-Key Cryptography





A one-way function is one that maps a domain into a range such that every function value has a unique inverse,





**🔍 Public-Key Cryptanalysis – Key Points**

1. **Brute-Force Attack:**
   * Public-key systems are vulnerable to brute-force like symmetric ones.
   * 🔐 **Solution**: Use **large keys**, but...
   * ⚠️ **Trade-off**: Larger keys → **slower** encryption/decryption due to higher computational complexity.
2. **Key Size Challenge:**
   * Must balance between:
     + ❌ Too small → vulnerable
     + ❌ Too large → impractical performance
   * 🛠️ Result: Public-key cryptography is mainly used for **key management** and **digital signatures**, not general data encryption.
3. **Private Key Recovery Attack:**
   * Attackers may try to derive **private key from public key**.
   * ❓ No public-key algorithm (even RSA) is **provably secure** against this.
4. **Probable-Message Attack:**
   * If message is predictable (e.g., a 56-bit DES key), attacker can:
     + Encrypt all possible messages.
     + Match with ciphertext to find the message.
   * 🔐 **Countermeasure**: Add **random bits** to the message to prevent matching.

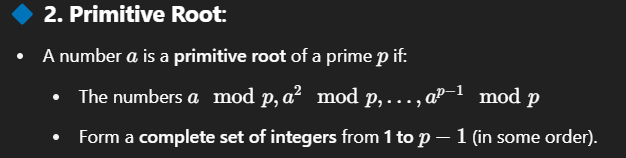
QP

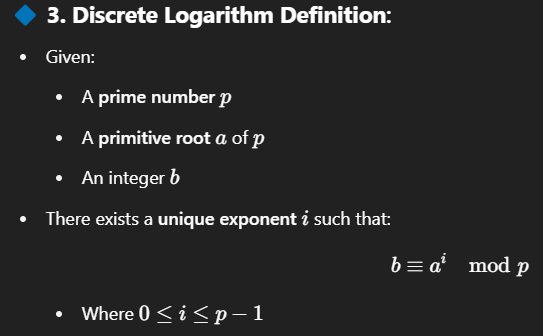
Explain Diffie-Hellman key exchange algorithm in detail. 08

Ans:

The purpose of the algorithm is to enable two users to securely exchange a key that can then be used for subsequent symmetric encryption of messages.

**Diffie-Hellman** algorithm relies on the **difficulty of computing discrete logarithms** for its security.

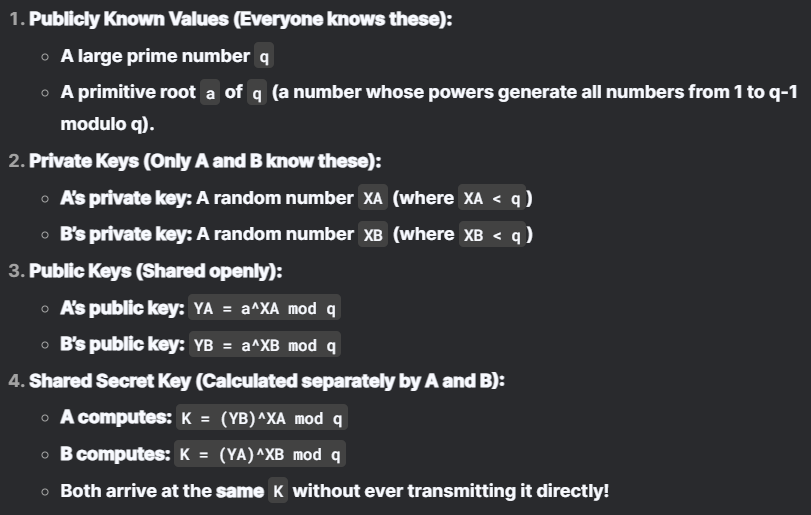


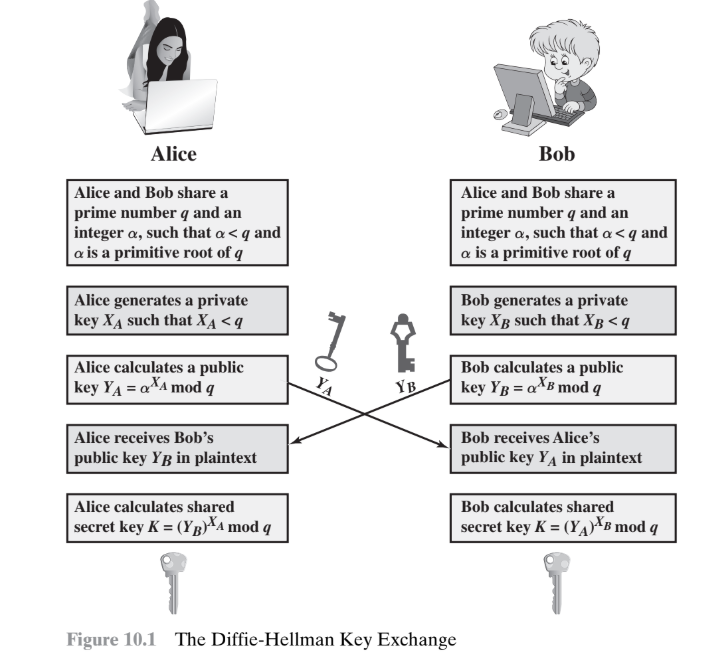


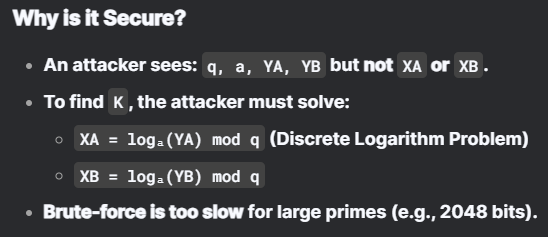
This exponent i is called the **discrete logarithm** of b to the base a modulo p.

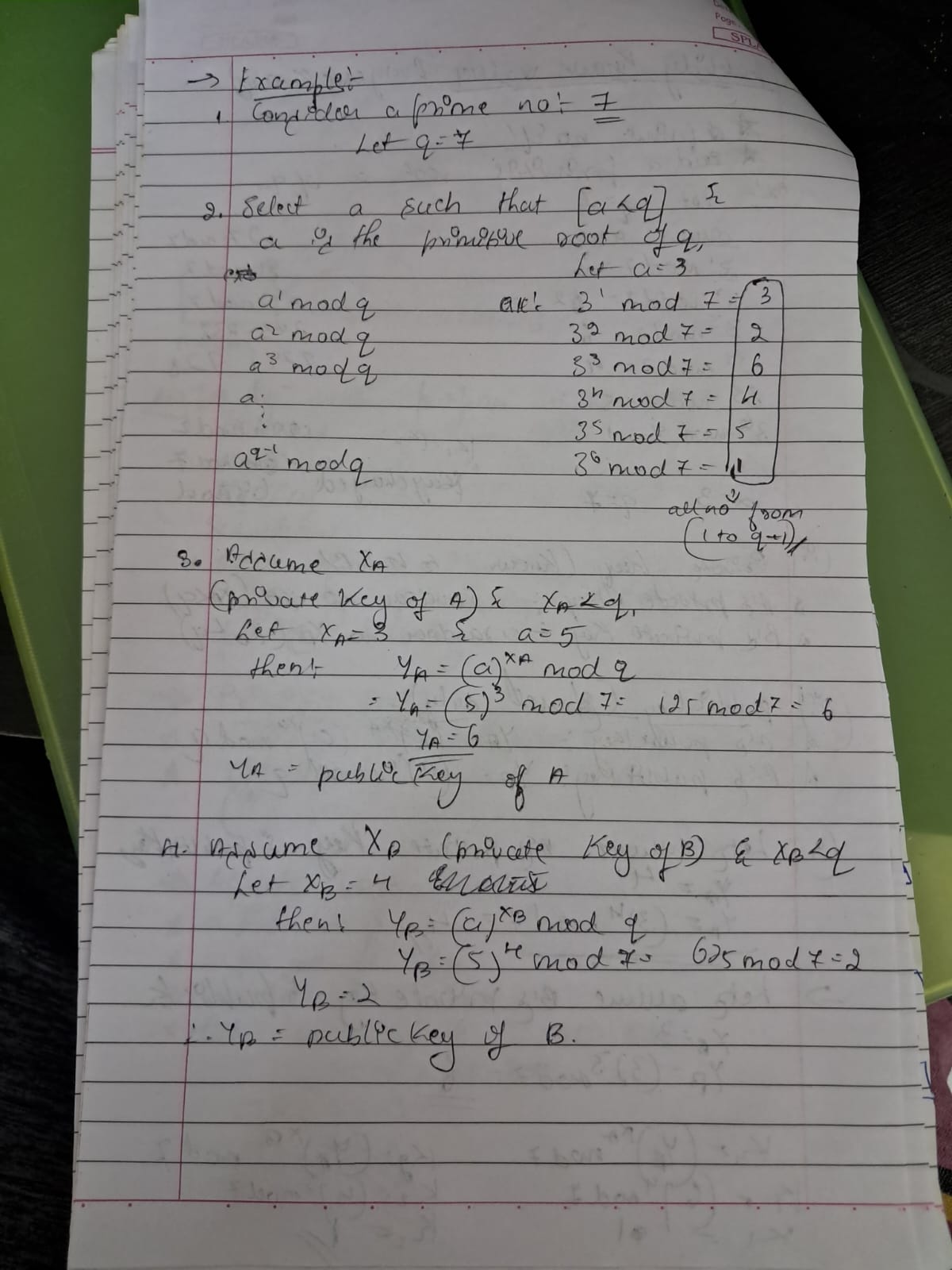
Algorithm:

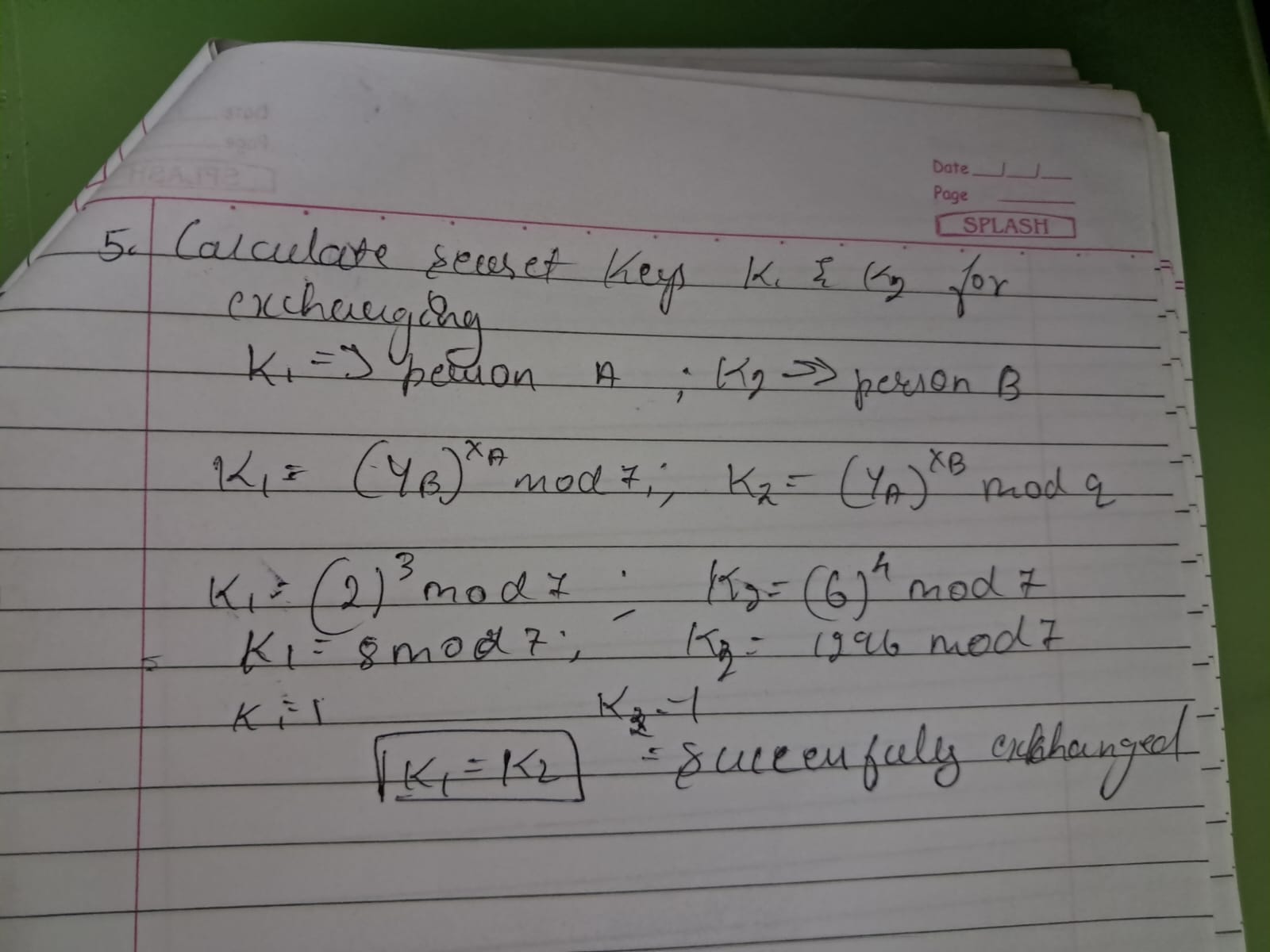
Two parties (A and B) want to securely exchange a secret key over an insecure channel (like the internet) so they can use it for encrypted communication.











**Key Exchange Protocols Using Diffie-Hellman (Exam Summary)**

**1. Basic Diffie-Hellman Key Exchange (Direct Communication)**

* **Scenario:** User **A** wants to securely communicate with **B** using an encrypted connection.
* **Steps:**
  1. **A** generates a **one-time private key (XA)** and computes **YA = a^XA mod q**.
  2. **A** sends **YA** to **B**.
  3. **B** generates its own **private key (XB)** and computes **YB = a^XB mod q**.
  4. **B** sends **YB** to **A**.
  5. Both compute the **shared secret key**:
     + **A computes:** **K = (YB)^XA mod q**
     + **B computes:** **K = (YA)^XB mod q**
  6. Now, **A & B** can encrypt messages using **K**.
* **Public Values (q & a):**
  1. Can be **pre-shared** or sent by **A** in the first message.

**2. Group Key Exchange (LAN Users with Central Directory)**

* **Scenario:** Multiple users (e.g., on a LAN) want secure pairwise communication.
* **Setup:**
  + Each user **i** generates a **long-term private key (Xi)** and computes **Yi = a^Xi mod q**.
  + All **Yi**, along with **q & a**, are stored in a **trusted central directory**.
* **How User j Sends Encrypted Message to User i:**
  1. **j** retrieves **i’s public key (Yi)** from the directory.
  2. **j** computes the **shared key**: **K = (Yi)^Xj mod q**.
  3. **i** computes the same **K = (Yj)^Xi mod q** (since **Yj = a^Xj mod q**).
  4. **j** encrypts the message with **K** and sends it to **i**.
* **Security Properties:**
  + **Confidentiality:** Only **i & j** know **K**, so others can’t decrypt.
  + **Authentication:** Only **j** could have encrypted with **K**, so **i** knows the sender is **j**.
  + **Limitation:** Does **not prevent replay attacks** (attacker can resend old messages).

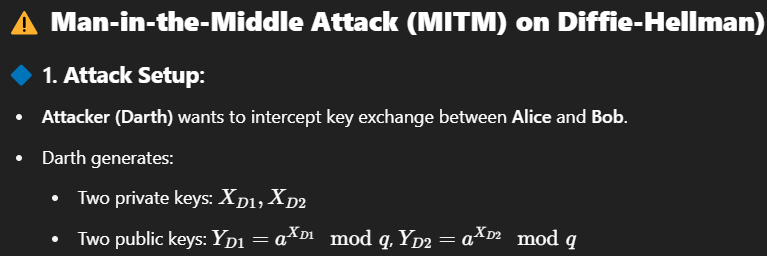
**Key Exam Points:**

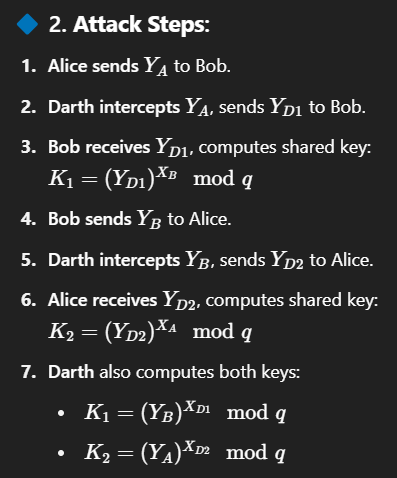
1. **Diffie-Hellman (DH) is used for:**
   * Secure **key exchange** (not encryption directly).
   * **One-time keys** (ephemeral) or **long-term keys** (static).
2. **Central Directory Use Case:**
   * Enables **any two users** to derive a shared key **without prior interaction**.
   * Provides **confidentiality + authentication** (but not replay protection).
3. **Security Limitations:**
   * **No replay protection** (attacker can reuse old messages).
   * **Man-in-the-middle (MITM) risk** if no additional authentication (e.g., digital signatures).

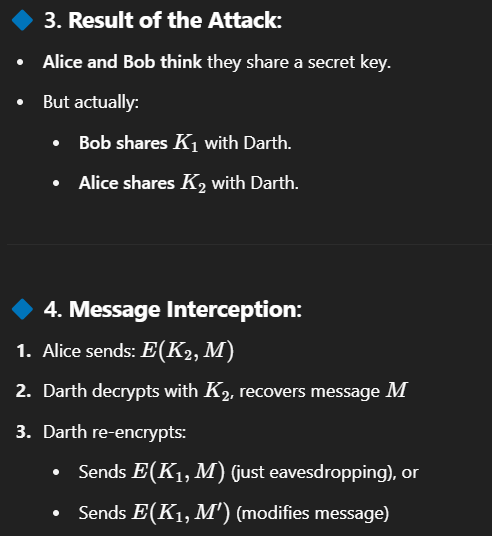
**Example (Exam Question Style):**

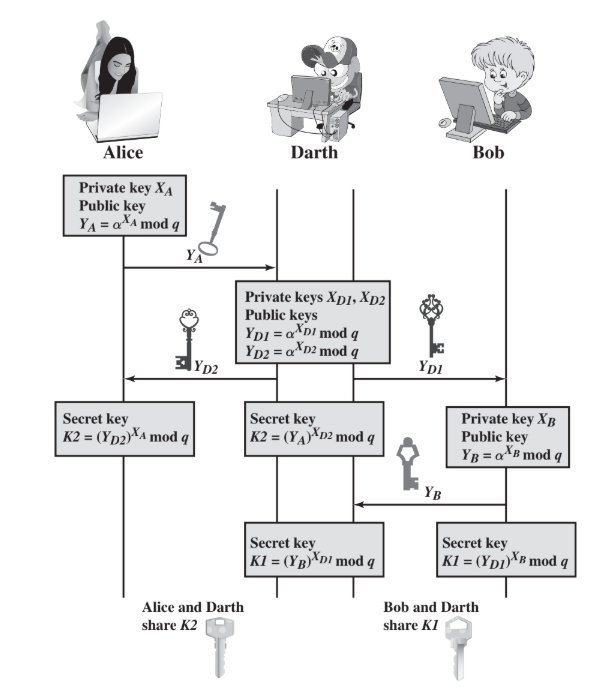
**Q:** In a Diffie-Hellman-based group key exchange, how does user **B** securely send a message to user **A**?  
**A:**

1. **B** retrieves **A’s public key (YA)** from the central directory.
2. **B** computes **K = (YA)^XB mod q**.
3. **A** computes the same **K = (YB)^XA mod q**.
4. **B** encrypts the message with **K** and sends it to **A**.
5. Only **A & B** know **K**, ensuring **confidentiality** and **authentication**.









**🔴 5. Why the Attack Works:**

* The protocol does **not authenticate** the users.

**✅ 6. Solution:**

* Use **digital signatures** and **public-key certificates** to authenticate participants and prevent MITM attacks.

**✅ Cryptographic Hash Function – Summary**

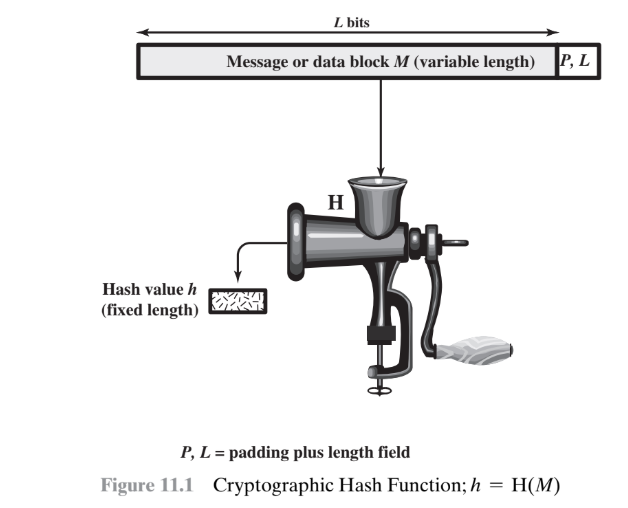
* A **hash function H** takes a variable-length input **M** and produces a **fixed-size output h = H(M)**.
* A **"good" hash function** produces outputs that appear **random** and are **evenly distributed**.
* **Main goal**: Ensure **data integrity** – even a small change in M causes a major change in h.

**🔐 Cryptographic Hash Function – Security Properties:**

1. **One-way property**:
   * It is **computationally infeasible** to find a message M that maps to a given hash h.
2. **Collision-free property**:
   * It is **very hard** to find **two different messages** M1 and M2 such that **H(M1) = H(M2)**.

**🧩 Working Principle:**

* Input message is **padded** to a fixed block size (e.g., 1024 bits).
* Padding includes the **length of the original message** (security measure).
* This makes it hard for attackers to forge messages with the same hash.



Let’s use the **SHA-256** hash function:

🔹 Input 1: M1 = "hello"  
H(M1) = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824

🔹 Input 2 (slightly changed): M2 = "hello!"  
H(M2) = 334d98e2f4b96d9d9f8a19c37e8fcabfdbdfdb396be4a80cc92f67e6b1af072a

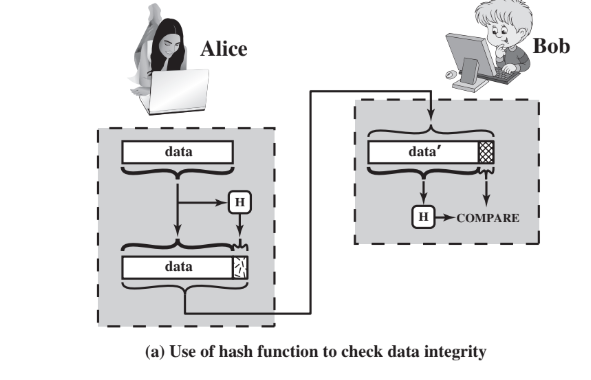
**Applications of the cryptographic hash function**

**Message authentication:**

a hash function is used to provide message authentication, the hash function value is often referred to as a **message digest.**

The essence of the use of a hash function for message authentication is as follows.

* The sender computes a hash value as a function of the bits in the message and transmits both the hash value and the message.
* The receiver performs the same hash calculation on the message bits and compares this value with the incoming hash value.
* If there is a mismatch, the receiver knows that the message (or possibly the hash value) has been altered



the hash function must be protected so that if an adversary alters or replaces the message

Alice transmits a data block and attaches a hash value. Darth intercepts the message, alters or replaces the data block, and calculates and attaches a new hash value. Bob receives the altered data with the new hash value and does not detect the change. To prevent this attack, the hash value generated by Alice must be protected.

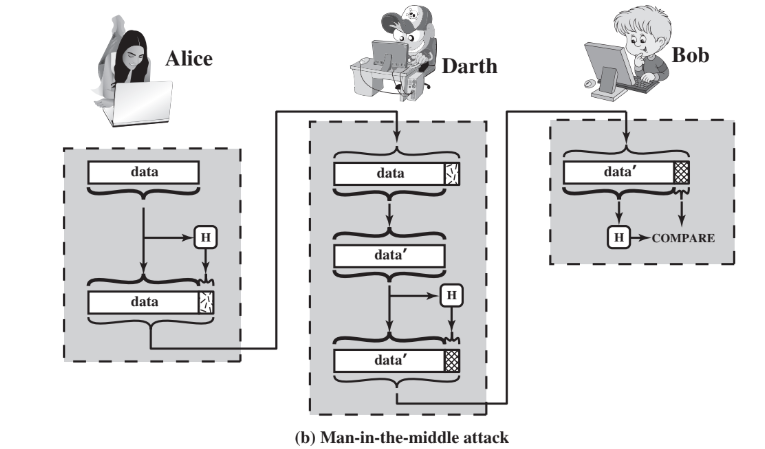
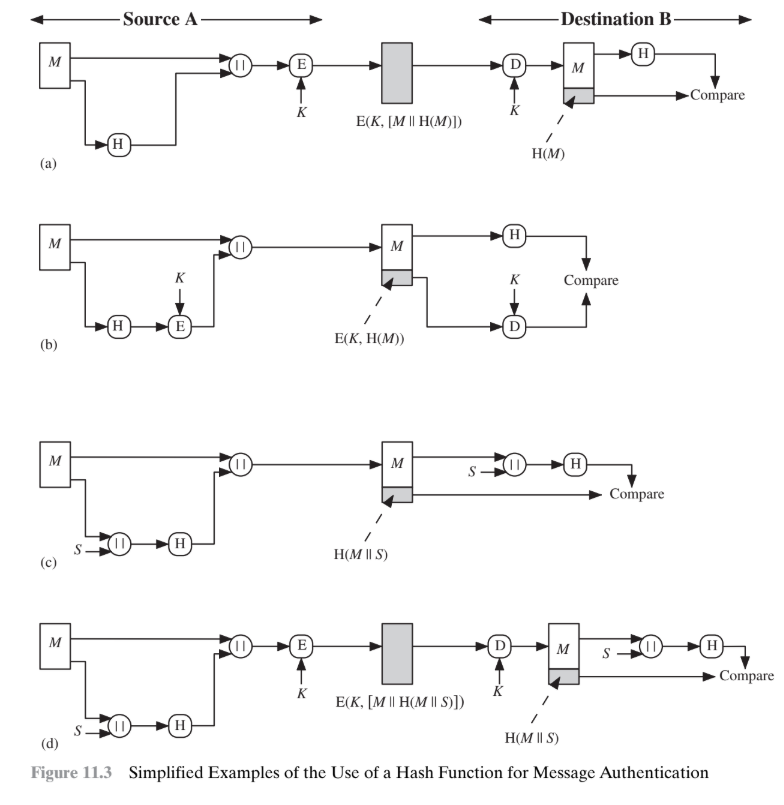


Figure 11.3 illustrates a variety of ways in which a hash code can be used to provide message authentication, as follows.



**🔐 Ways to Use Hash for Message Authentication:**

**a. Encrypt Message + Hash (Symmetric Key)**

* Both are encrypted.
* Provides **authentication** and **confidentiality**.
* Only sender and receiver know the key.

**b. Encrypt Only the Hash**

* Message is sent as-is, only hash is encrypted.
* **Authentication only**, faster than (a).
* Used when **confidentiality isn’t required**.

**c. Use Hash + Shared Secret (No Encryption)**

* Hash is computed using **message + secret key (S)**.
* Only sender and receiver know S, so attacker can’t fake it.
* **No encryption**, but ensures **integrity and authenticity**.

**d. Encrypt Message + Hash from (c)**

* Adds **confidentiality** to method (c).
* Ensures **authenticity, integrity, and confidentiality**.

When **confidentiality isn’t needed**, **method (b)** (encrypting only the hash) is **better** than (a) or (d) because it needs **less computation**.

There’s also growing interest in avoiding encryption entirely (like in method c) because:

**🔍 Reasons to Avoid Full Encryption:**

1. **Slow software** – Encrypting messages takes time, especially if messages come rapidly.
2. **Hardware cost** – Encryption chips (e.g., DES) cost money, and adding them to every device increases cost.
3. **Inefficiency for small data** – Encryption hardware works best on large data; for small data, setup time is too high.
4. **Licensing issues** – Some encryption methods are patented and require payment to use.

**MAC**

Message authentication is commonly done using a **Message Authentication Code (MAC)**, which is a **keyed hash function**.

* Both sender and receiver share a **secret key**.
* The **MAC function** takes the key and message, then creates a hash called the **MAC value**.
* The receiver uses the same key to verify the MAC.
* An attacker **cannot forge** the MAC without knowing the key.
* MAC = E(K, H(M)), a secure combination of hashing and encryption.
* Special MAC algorithms are faster than general encryption.

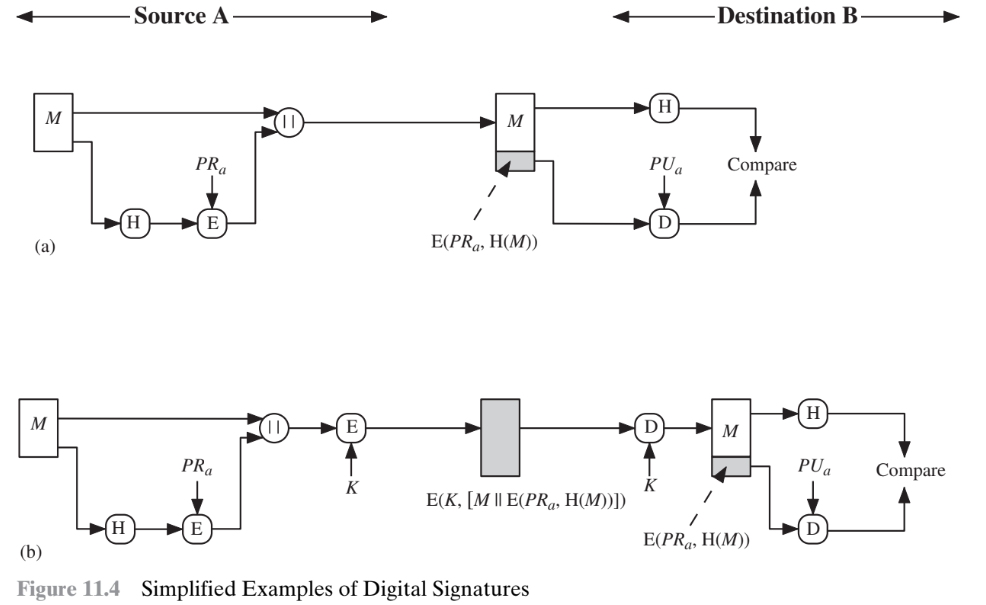
Digital Signature

**Digital Signature** is like a **secure stamp** that proves the sender's identity and message integrity.

* A **hash of the message** is encrypted with the **sender’s private key**.
* Anyone with the **sender’s public key** can verify the signature and ensure the message hasn’t changed.
* This ensures **authentication**, **integrity**, and **non-repudiation** (the sender can't deny sending it).

**Two methods:**

1. **Digital Signature only**: Encrypt the hash with sender’s private key → proves sender identity.
2. **Digital Signature + Confidentiality**: Encrypt message + signed hash with symmetric key → hides message content too.



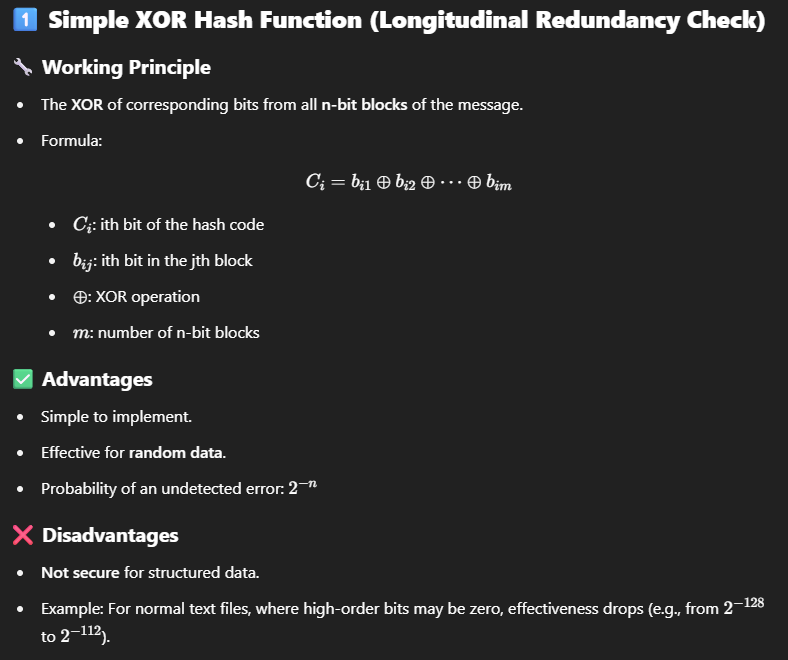
This is different from MACs, which use a **shared secret key**.  
Digital signatures use **public-key cryptography**, so anyone can verify, but only the real sender can sign.

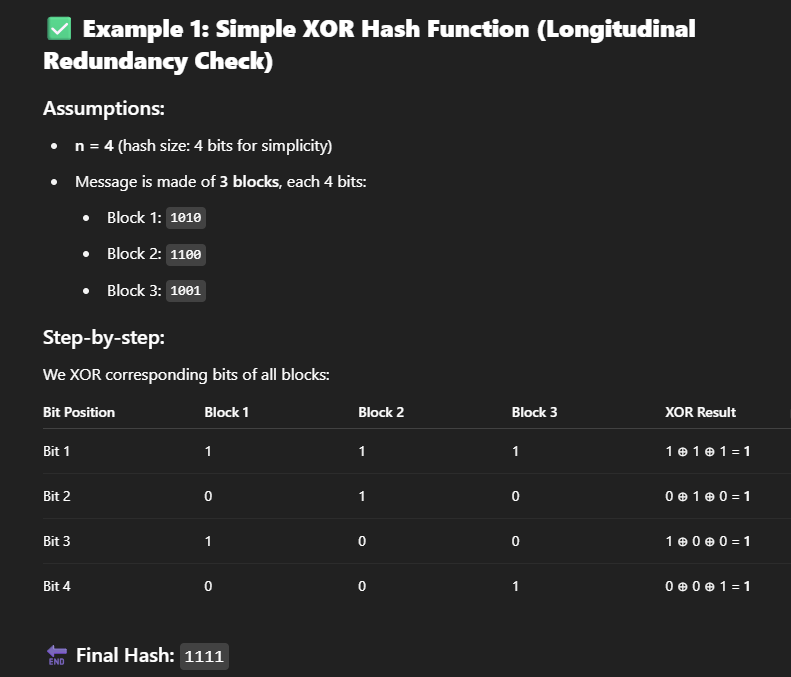
**🔐 Other Applications of Hash Functions – Key Points**

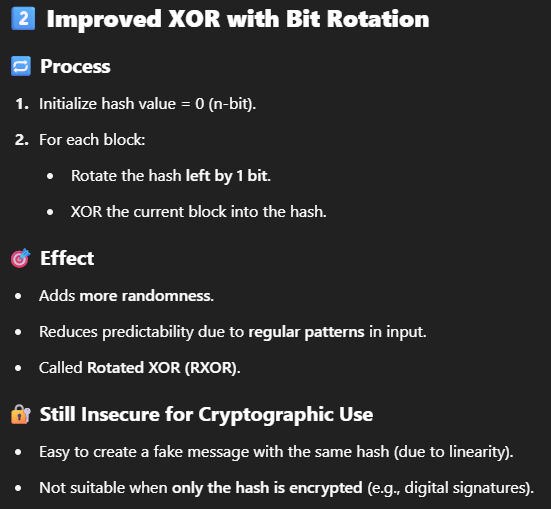
1. **Password Protection:**
   * Operating systems store **hashes of passwords**, not the passwords themselves.
   * During login, system compares **entered password's hash** with the **stored hash**.
   * ✅ Enhances security—actual passwords can’t be retrieved by hackers.
2. **Intrusion & Virus Detection:**
   * Store hash H(F) for each file and protect it (e.g., on secure CD/DVD).
   * Later, recompute H(F) and compare.
   * ✅ Detects if a file was modified—tampering changes the hash.
3. **Pseudorandom Function (PRF) & Number Generator (PRNG):**
   * Cryptographic hash functions help generate **pseudorandom outputs**.
   * 📌 Commonly used for **symmetric key generation** (explained in Chapter 12).

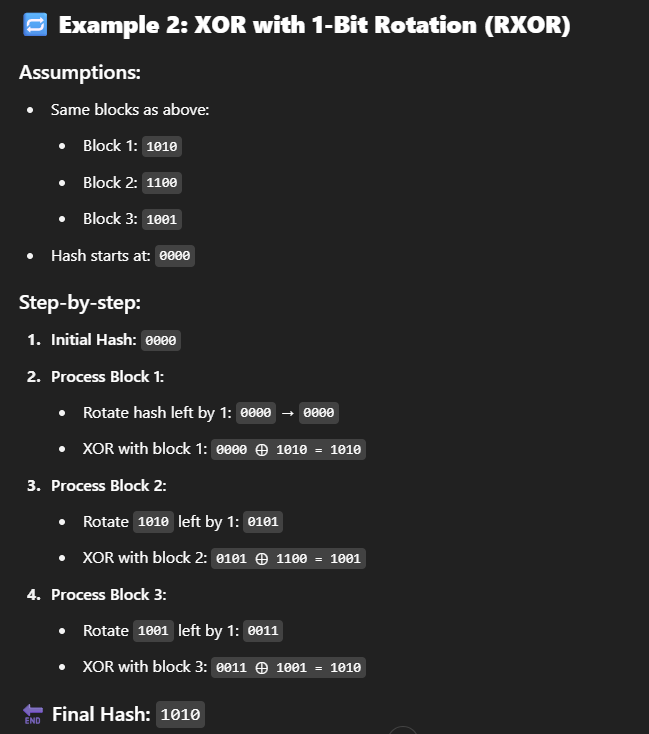
Two Simple Hash Function

The input (message, file,etc.) is viewed as a sequence of n-bit blocks. The input is processed one block at a time in an iterative fashion to produce an n-bit hash function.

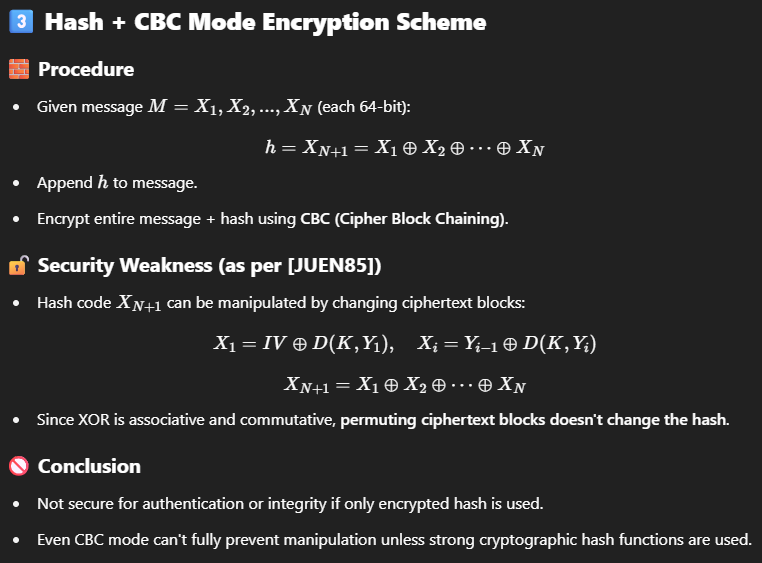






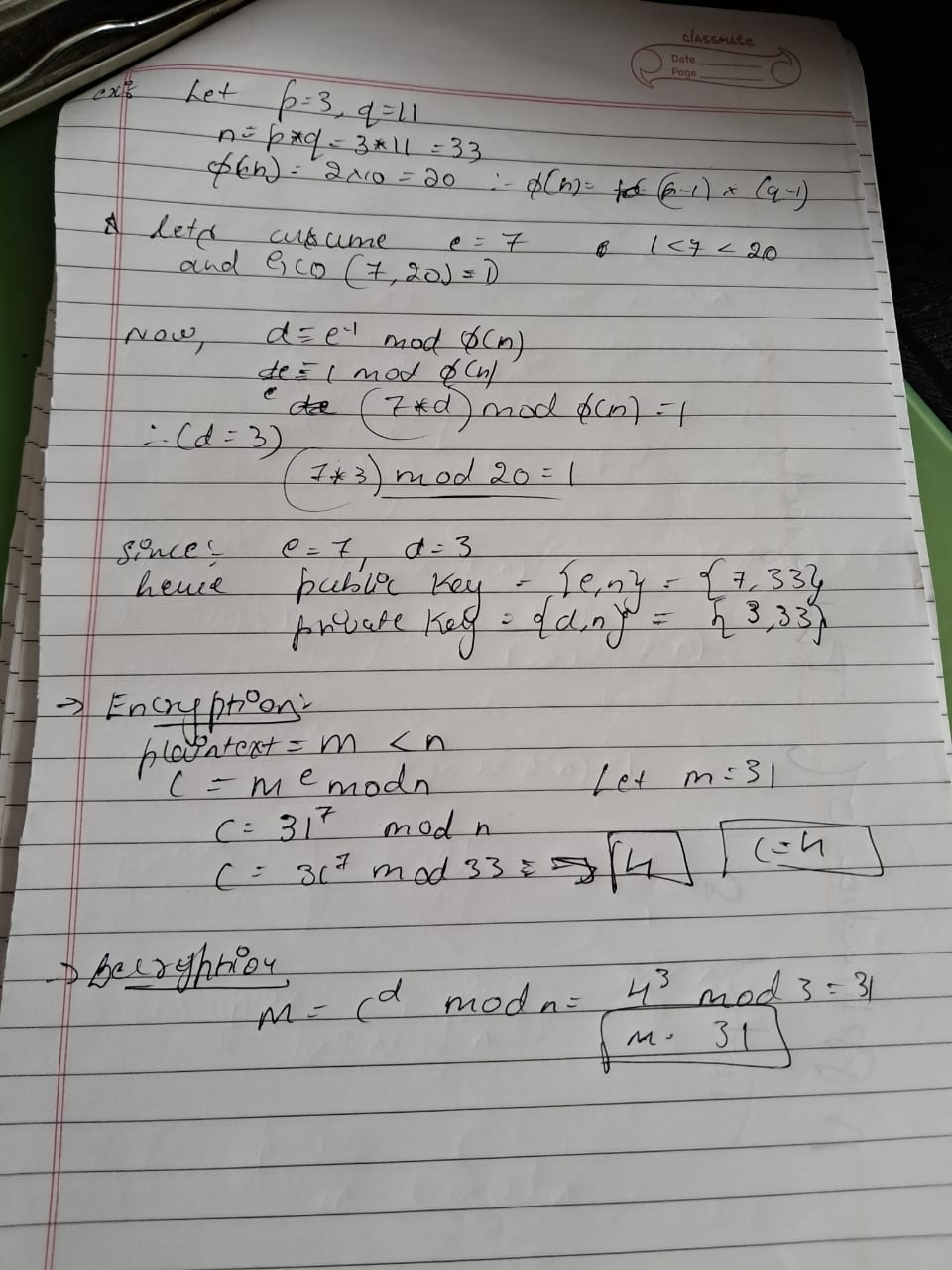


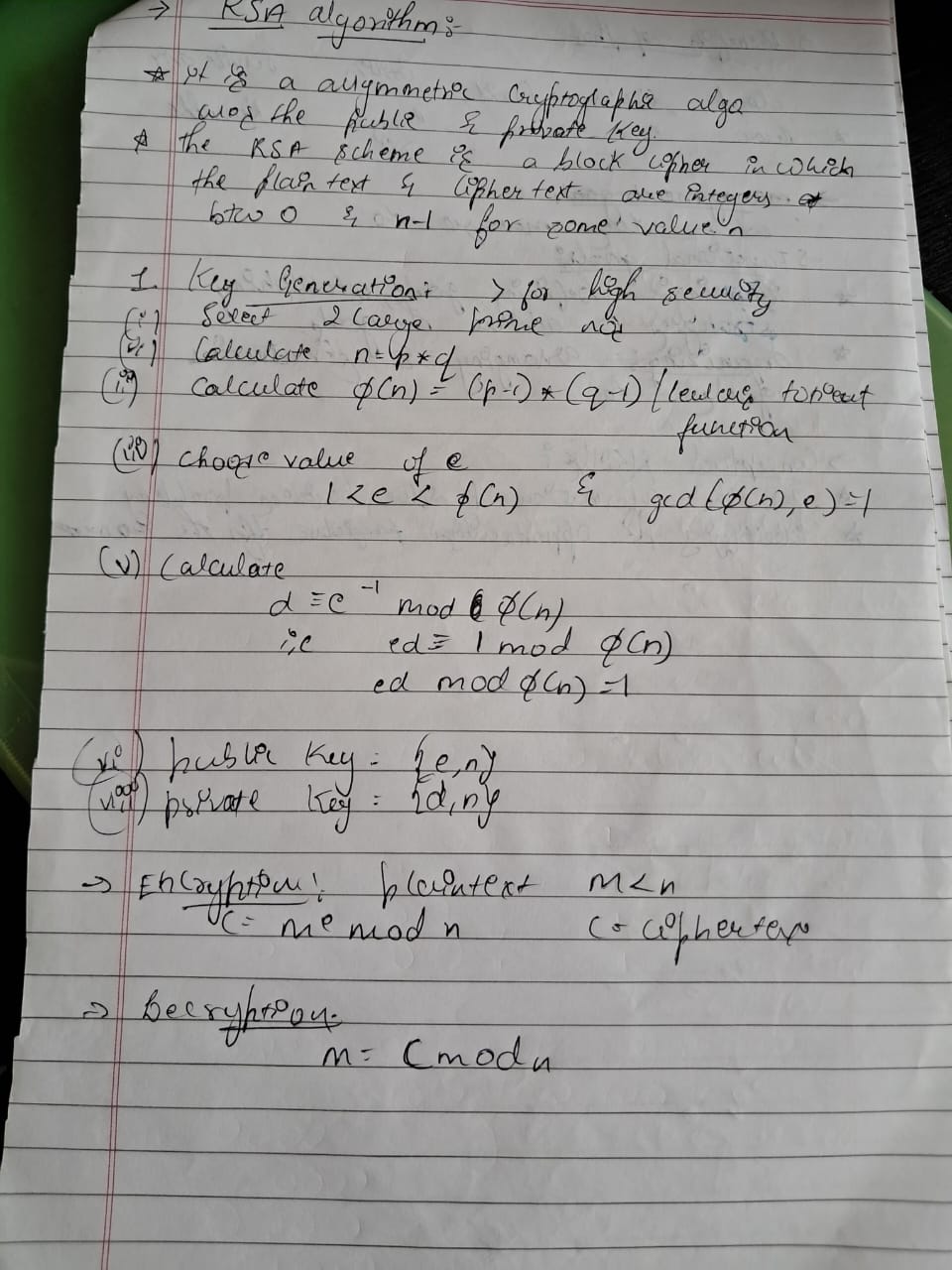
A technique originally proposed by the National Bureau of Standards used the simple XOR applied to 64-bit blocks of the message and then an encryption of the entire message that used the cipher block chaining (CBC) mode. We can define the scheme as follows: Given a message M consisting of a sequence of 64-bit blocks X1, X2, c, XN, define the hash code h = H(M) as the block-by- block XOR of all blocks and append the hash code as the final block:



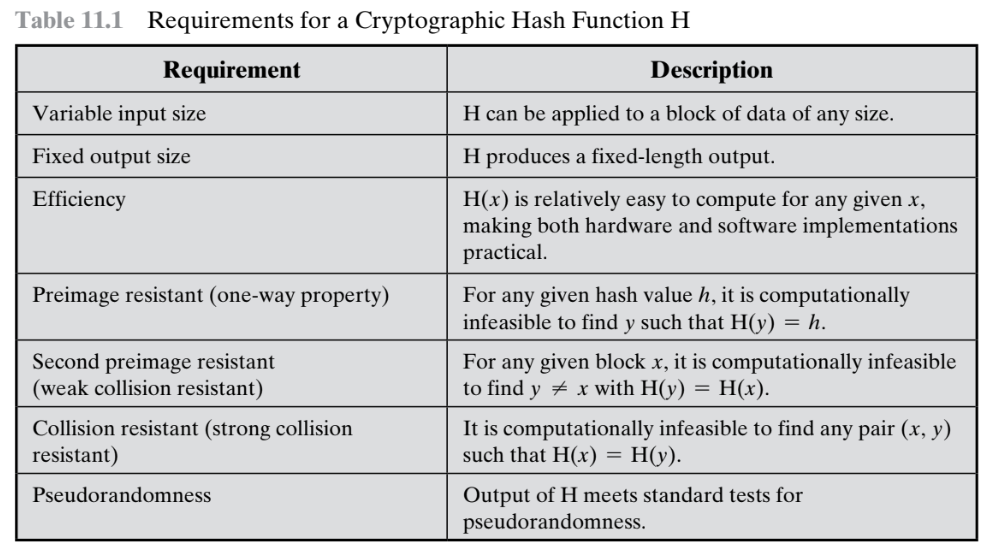
IV = initialization vector used only for the frist msg..

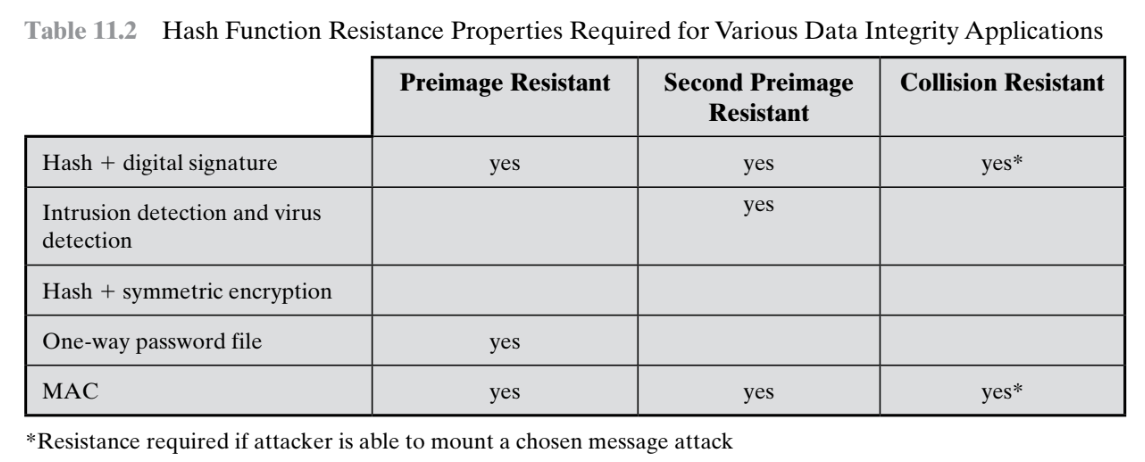
RSA





Security Requirements for Cryptographic Hash Functions





Attacks on the Hash Function  
**🔐 What Are the Two Main Types of Attacks on Hash Functions?**

1. **Brute-force attacks**  
   → Try all possible inputs randomly until something works.  
   → Doesn’t care **how the hash function works** — just relies on its **bit length**.
2. **Cryptanalysis attacks**  
   → Try to break the hash function by studying its **design or weaknesses**.

🧱 Types of Brute-Force Attacks on Hash Functions

**1️ Preimage Attack**

**Goal:** Given a hash value h, find any message y such that H(y) = h.

This is like trying to **reverse the hash**, which should be very hard.

🧠 **Effort required:**  
If the hash size is m bits, the attacker would need about 2^m tries (on average 2^(m-1)).

📦 Example:  
If m = 256 (as in SHA-256), it would take about 2^256 tries — practically **impossible**.

**2️ Second Preimage Attack**

**Goal:** Given a message x, find a different message y ≠ x such that H(y) = H(x).

This is like **faking another message** with the **same hash**.

🧠 **Effort required:**  
Still about 2^m tries (same as preimage attack).

**3️ Collision Attack**

**Goal:** Find **any two different messages** x and y such that H(x) = H(y).

This is **easier** than the other two, thanks to the **Birthday Paradox**.

**🎉 The Birthday Paradox (in Simple Terms)**

It says:

In a group of just 23 people, there’s more than a 50% chance **two people share a birthday**.

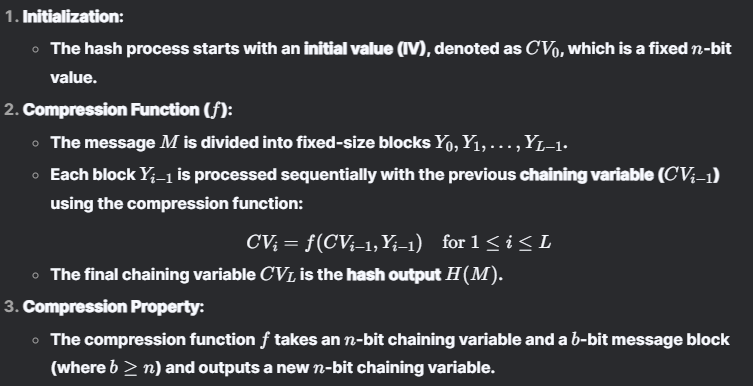
In hashing terms:

You only need about 2^(m/2) random messages to **likely** find a collision.

🧠 **Effort required:**

* For a 256-bit hash, only need about 2^128 tries to find **a collision**.
* Still very hard — but **easier than 2^256** (preimage/second-preimage).

Yuval proposed the following strategy to exploit the birthday paradox in a collision resistant attack [YUVA79].

Crytpanalysis Attacks  


**👨‍🔬 How Do Cryptanalysts Attack It?**

They:

1. Focus on the **compression function f()**
2. Try to find **two inputs** that give the **same output** (i.e., a collision)
3. Work with knowledge of the **IV** (initial value)
4. Analyze how bits **change across rounds** inside f()

Similar to how people attack symmetric ciphers like AES or DES — by studying how data transforms step-by-step.

**🔁 Why Collisions Must Exist (But Should Be Hard to Find)**

* If a hash maps **very long messages** (e.g., 512+ bits) into **short hashes** (e.g., 256 bits),  
  then **collisions must exist** (pigeonhole principle).
* But it should be **computationally infeasible** to find those collisions.