Reading for L3

▼ Table of Content

2.3: public-key encryption

Public-Key Encryption Structure

Video: Asymmetric Encryption - Simply explained

Public-key encryption, first proposed by Diffie and Hellman in 1976, is a revolutionary encryption method that uses **two separate keys**: a **public key** (for encryption) and a **private key** (for decryption). This **asymmetric** system contrasts with symmetric encryption, which uses the same key for both encryption and decryption. Public-key encryption has major implications for **confidentiality**, **key distribution**, and **authentication**.

Key Points:

1. **Misconceptions**:

- **Public-key encryption is not inherently more secure** than symmetric encryption. Both types of encryption depend on the **key length** and the **computational difficulty** of breaking the cipher.
- **Public-key encryption has not made symmetric encryption obsolete**. Its computational overhead makes symmetric encryption still essential.
- **Key distribution is not simpler** in public-key encryption. Although public keys are publicly shared, protocols are still needed for secure distribution, often involving central agents.

2. **Basic Elements of Public-key Encryption**:
   - **Plaintext**: The readable message.
   - **Encryption Algorithm**: Transforms plaintext using a key.
   - **Public and Private Keys**: A pair of keys, one for encryption and the other for decryption.
   - **Ciphertext**: The encrypted message, which depends on both the plaintext and key.
   - **Decryption Algorithm**: Recovers the plaintext from ciphertext using the matching private key.

   The public key is available to everyone, while the private key is kept secret by its owner.

3. **Process**:
   - **Key Generation**: Each user generates a key pair—public and private.
   - **Public Key Sharing**: The public key is made available in a public register.
   - **Encryption**: If Bob wants to send Alice a private message, he encrypts it with Alice's public key.
   - **Decryption**: Alice decrypts the message using her private key.

   This system allows secure communication since the private key never needs to be distributed. If Alice's private key is kept secure, no one else can decrypt the message.

4. **Modes of Operation**:

- **Confidentiality** (Figure 2.6a): If Bob encrypts a message using Alice's public key, only Alice can decrypt it with her private key, ensuring confidentiality.
- **Authentication/Data Integrity** (Figure 2.6b): If Bob encrypts a message using his private key, anyone with his public key can decrypt it, proving the message's authenticity and ensuring it hasn't been tampered with.

The effectiveness of public-key encryption depends on the **security of the keys**, the **algorithm**, and the **protocols** used for key management and communication.

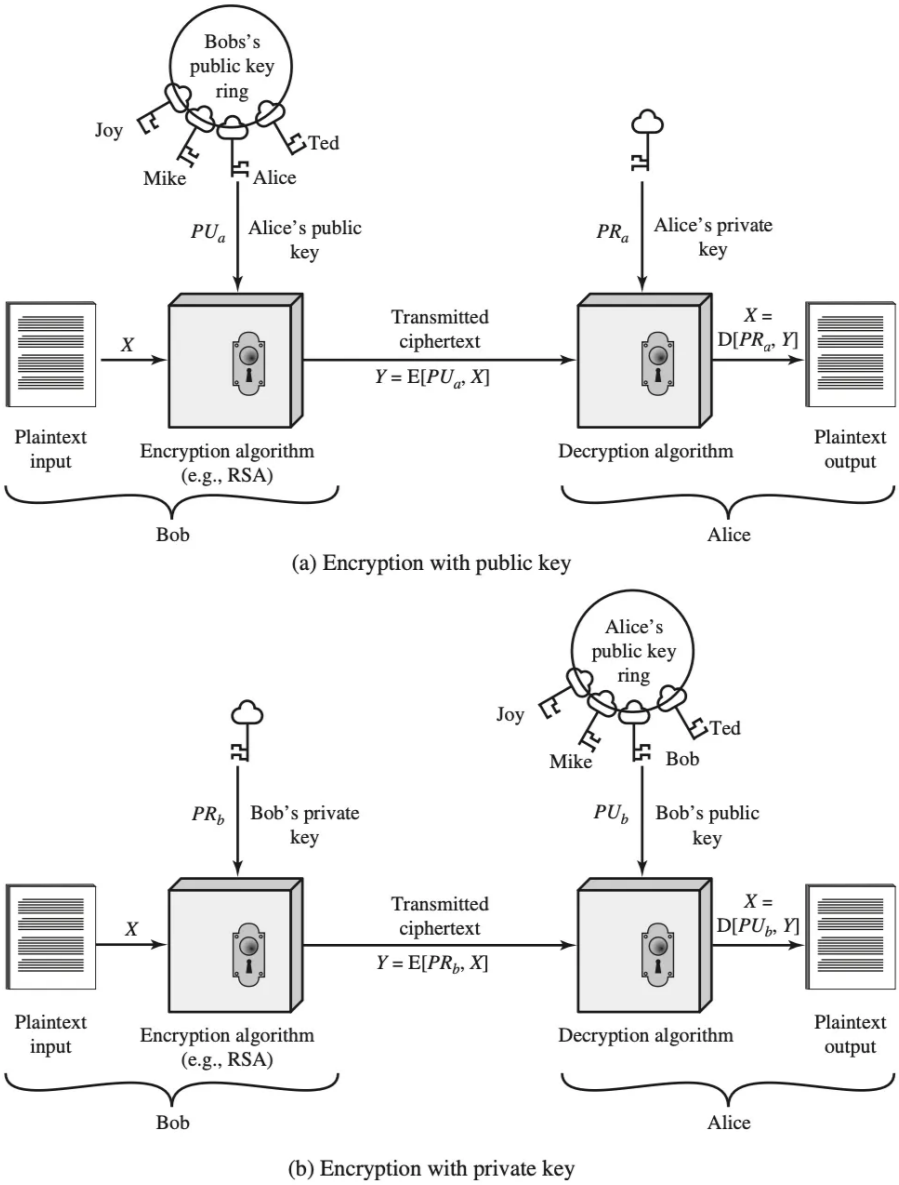


Figure 2.6 **Public-Key Cryptography**

Applications for Public-Key Cryptosystems

Public-key systems use two cryptographic keys: a **private key** (kept secret) and a **public key** (shared openly). Depending on the application, the sender may use their **private key**, the receiver's **public key**, or both to perform cryptographic functions. Public-key cryptosystems are typically used in three main areas:

1. **Digital Signatures**
2. **Symmetric Key Distribution**
3. **Encryption of Secret Keys**

Different algorithms are designed to support one or more of these applications. Some algorithms can be used for all three purposes, while others are limited to one or two. The applications of the algorithms are detailed in **Table 2.3**.

Table 2.3 Applications for Public-Key Cryptosystems

Algorithm	Digital Signature	Symmetric Key Distribution	Encryption of Secret Keys
RSA	Yes	Yes	Yes
Diffie–Hellman	No	Yes	No
DSS	Yes	No	No
Elliptic Curve	Yes	Yes	Yes

Requirements for Public-Key Cryptography

The **cryptosystem** proposed by Diffie and Hellman relies on two related keys: a **public key** and a **private key**. They outlined the following conditions for such a system:

1. **Key generation**: It should be easy for a party *B* to generate a public-private key pair (*PUb*,*PRb*).
2. **Encryption**: A sender *A*, knowing the public key *PUb* and the message *M* , can easily encrypt the message into ciphertext *C* using the public key

3. **Decryption**: The receiver *B* can easily decrypt the ciphertext *C* using their private key *PRb* to recover the original message *M*.
4. **Private key security**: It should be computationally infeasible for an opponent, knowing the public key, *PUb*, to determine the corresponding private key, *PRb*.
5. **Message security**: It should be computationally infeasible for an opponent to recover the original message *M* from the public key and the ciphertext.
6. **Reversible key use**: Either of the two related keys (public or private) can be used for encryption, with the other key used for decryption.

Asymmetric Encryption Algorithms

1. **RSA (Rivest-Shamir-Adleman)**: Developed in 1977, RSA is one of the first public-key encryption schemes and remains widely used today. It operates as a block cipher, with plaintext and ciphertext as integers. RSA gained fame in 1977 when its inventors challenged readers to decrypt a cipher, which was later solved in 1994 using significant computational power. RSA's key size has increased over time, with 1024-bit keys currently considered secure for most applications.
2. **Diffie-Hellman Key Exchange**: Introduced in 1976, the Diffie-Hellman algorithm allows two users to securely agree on a shared secret key for symmetric encryption. It is used primarily for key exchange rather than encryption or digital signatures.
3. **Digital Signature Standard (DSS)**: Introduced by NIST in 1994, the DSS uses the Digital Signature Algorithm (DSA) and SHA-1 to provide a method for creating digital signatures. Unlike RSA, DSA is limited to digital signatures and cannot be used for encryption or key exchange. The DSS has undergone several revisions since its introduction.
4. **Elliptic Curve Cryptography (ECC)**: ECC is emerging as a competitor to RSA due to its ability to provide the same level of security with smaller key sizes, reducing processing overhead. While ECC is gaining traction in standards and products, its confidence level is not yet as high as RSA due to less extensive cryptanalysis.

2.4: DIGITAL SIGNATURES AND KEY MANAGEMENT

Digital Signature

Public-key encryption can be used for **authentication** through a **digital signature**, which ensures **origin authentication**, **data integrity**, and **non-repudiation**. A digital signature is a unique cryptographic transformation of data that proves a message's authenticity, verifies it hasn't been altered, and prevents the signer from denying the signature.

According to **FIPS PUB 186-4** (Digital Signature Standard), there are three main digital signature algorithms:

1. **Digital Signature Algorithm (DSA)**: Based on the difficulty of discrete logarithms.
2. **RSA Digital Signature Algorithm**: Based on the RSA public-key algorithm.
3. **Elliptic Curve Digital Signature Algorithm (ECDSA)**: Based on elliptic curve cryptography.

Digital Signature Process:

- **Bob** sends a message to **Alice**.
- Bob uses a **secure hash function** (like SHA-512) to generate a hash of the message, then signs the hash with his **private key** to create a digital signature.
- Bob sends the message along with the signature to Alice.
- **Alice** calculates the hash of the received message and verifies the signature using Bob’s **public key**.
- If the signature is valid, Alice is assured that the message came from Bob, hasn't been altered, and wasn't signed by anyone else (as only Bob has the private key).

Note: A digital signature does **not** provide **confidentiality**. The message can still be seen by anyone, but it cannot be altered without invalidating the signature.

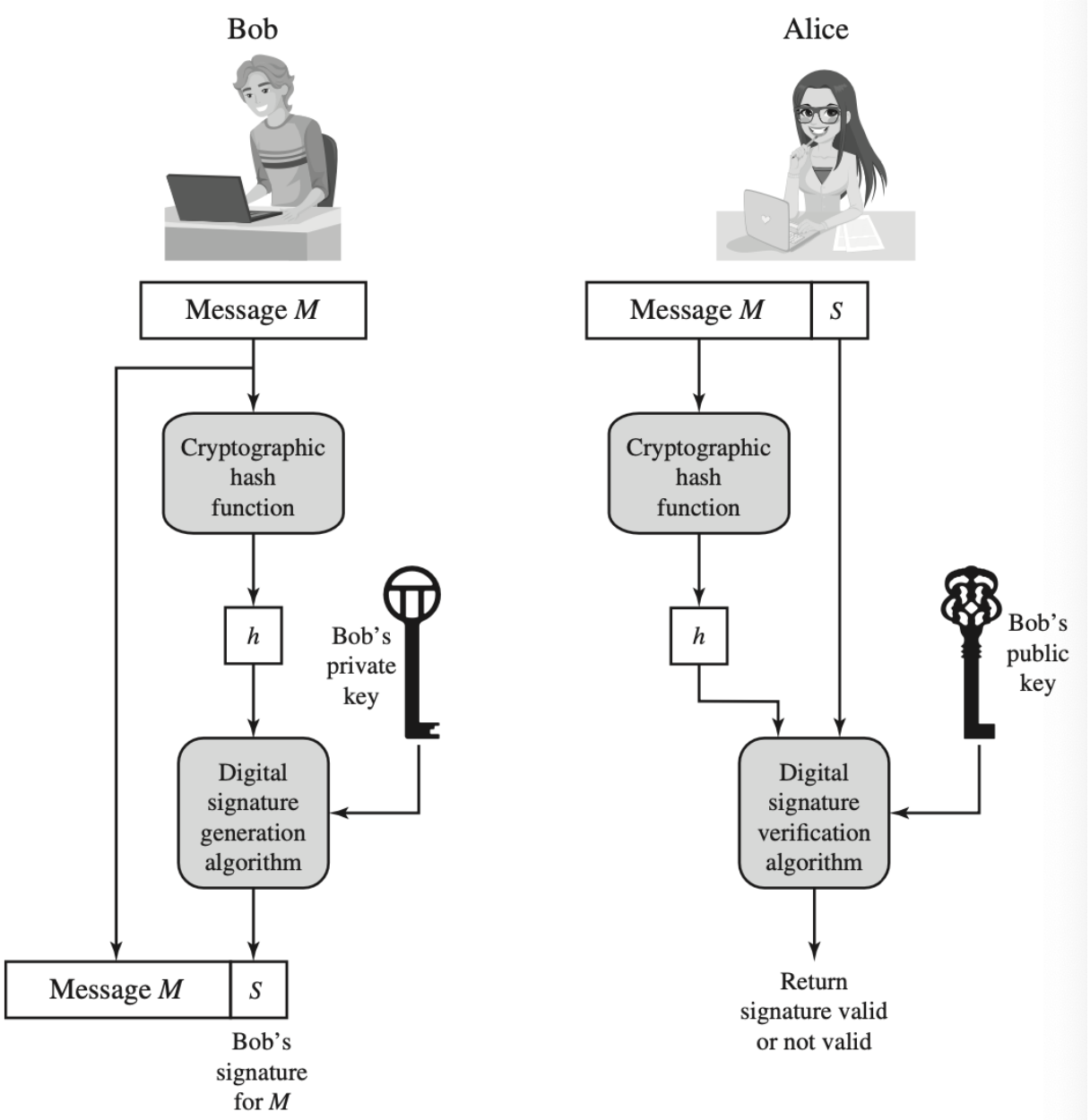


Figure 2.7 **Simplified Depiction of Essential Elements of Digital Signature Process**

Public-Key Certificates

Video: What are Digital Signatures? - Computerphile

Public-key encryption relies on the idea that public keys are openly shared, but this creates a vulnerability: anyone can forge a public key and impersonate another user, potentially intercepting messages or authenticating as someone else. To solve this, **public-key certificates** are used.

A public-key certificate consists of a **user's public key**, their **user ID**, and is **signed by a trusted third party**, such as a **Certificate Authority (CA)**. This ensures the legitimacy of the public key and prevents forgery. The CA's signature verifies the public key's authenticity, and the certificate includes the CA's details and its validity period.

Key Steps:

1. **User generates a key pair**: public and private keys.
2. The user creates an **unsigned certificate** with their public key and user ID.
3. The user submits this certificate securely to a **CA**.
4. The CA uses a **hash function** to generate a hash of the certificate and signs it using the CA's private key.
5. The CA returns the **signed certificate** to the user.
6. The user can then **publish** the signed certificate.
7. **Others** can verify the certificate's validity by calculating the hash and verifying the CA's signature with the CA's public key.

The most common format for these certificates is the **X.509 standard**, used in protocols like **TLS**, **IPsec**, and **S/MIME**.

Symmetric Key Exchange Using Public-Key Encryption

In symmetric encryption, the two parties need to share a secret key to communicate securely. However, sharing this key becomes problematic when they are far apart. For example, if Bob wants to securely exchange messages with Alice, they both need a method to share the secret key without it being intercepted. If Alice is nearby, Bob could hand her the key directly, but if she is far away, he would face the issue of needing a secure way to send the key.

One solution is **Diffie-Hellman key exchange**, which allows Bob and Alice to agree on a shared secret key over an insecure channel. However, the basic version of Diffie-Hellman doesn't provide authentication, meaning that it can't confirm the identities of the communicating parties. There are variations of Diffie-Hellman and other public-key protocols that can address this authentication issue, ensuring that the key exchange is both secure and verified.

Digital Envelopes

A **digital envelope** is a method that combines public-key and symmetric encryption to securely send a message without needing the sender and receiver to share a symmetric key in advance. Here's how it works:

1. Bob prepares the message he wants to send.
2. He generates a **random symmetric key** for one-time use.
3. Bob encrypts the message using this one-time symmetric key.
4. He then encrypts the one-time symmetric key using **Alice's public key**.
5. Bob sends both the encrypted message and the encrypted key to Alice.

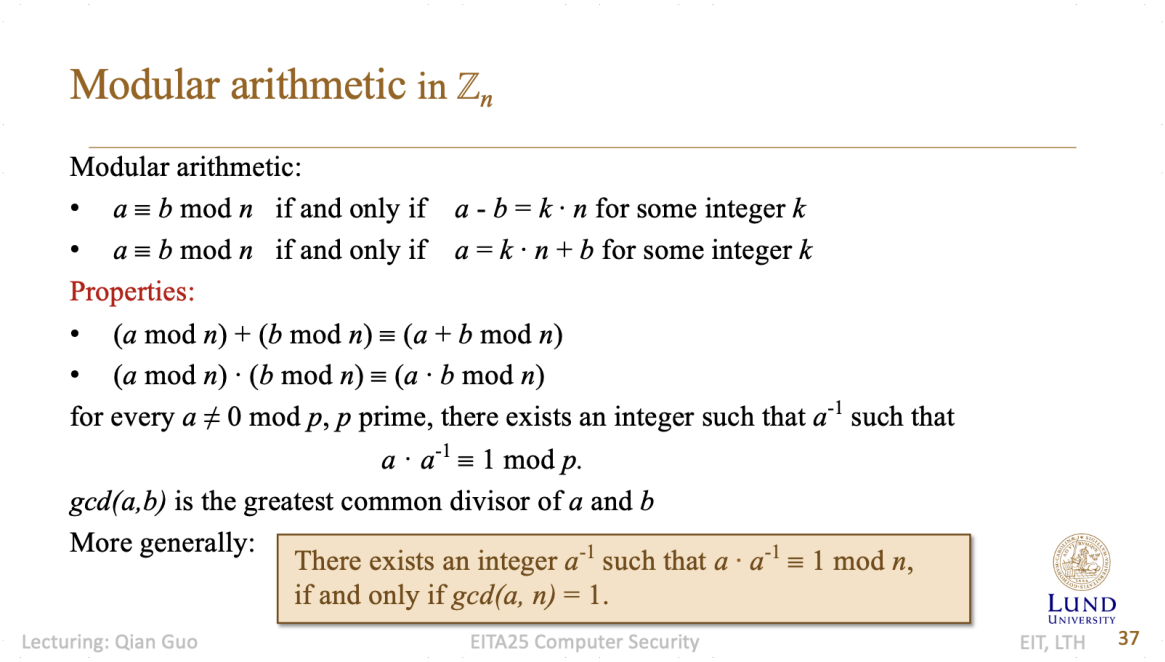
Only Alice, who holds the corresponding private key, can decrypt the one-time key and, subsequently, the message. If Bob has verified Alice’s public key via a certificate, he can be confident that the key is legitimate. This approach allows secure communication without the need for pre-shared secret keys.

21.4: THE RSA PUBLIC-KEY ENCRYPTION ALGORITHM

Prerequisite:

▼ Modular arithmetic

Video: What does a ≡ b (mod n) mean? Basic Modular Arithmetic, Congruence



Basic Concept:

When you perform modular arithmetic, you're working with **remainders**. For example, when you divide one number by another, the remainder of that division can be the result of a modular operation.

The notation for modular arithmetic is:

a mod n

This means the remainder when **a** is divided by **n**.

How It Works:

1. **Division with Remainder**: Take any number **a** and divide it by a positive number **n**. The result is the remainder, which is what modular arithmetic is all about.
2. **Example**:
   - Let's calculate .

     17 mod 5
     - Divide 17 by 5: remainder .

       17÷5=3 remainder 2

- So, $17 \bmod 5 = 2$

3. **More Examples**:
   - $9 \bmod 4$:
     - Divide 9 by 4: remainder .

       $9 \div 4 = 2$ remainder 1
     - So, $9 \bmod 4 = 1$
   - $-5 \bmod 3$:
     - Divide -5 by 3: The closest integer division is (because you're rounding towards zero).

       $-5 \div 3 = -2$
     - $-5 - (-2 \times 3) = -5 - (-6) = 1$
     - So, (the result is always positive in modular arithmetic).

       $-5 \bmod 3 = 1$

Properties of Modular Arithmetic:

1. **Additive Property**:

   $(a+b) \bmod n = (a \bmod n) + (b \bmod n)] \bmod n$

   Example:

   $(7+5) \bmod 3 = 12 \bmod 3 = 0$

   This is the same as:

   $(7 \bmod 3) + (5 \bmod 3) = 1+2 = 3 \bmod 3 = 0$

2. **Multiplicative Property**:

   $(a \times b) \bmod n = [(a \bmod n) \times (b \bmod n)] \bmod n$

   Example:

   $(4 \times 6) \bmod 5 = 24 \bmod 5 = 4$

   This is the same as:

   $(4 \bmod 5) \times (6 \bmod 5) = 4 \times 1 = 4 \bmod 5 = 4$

3. **Distributive Property**:

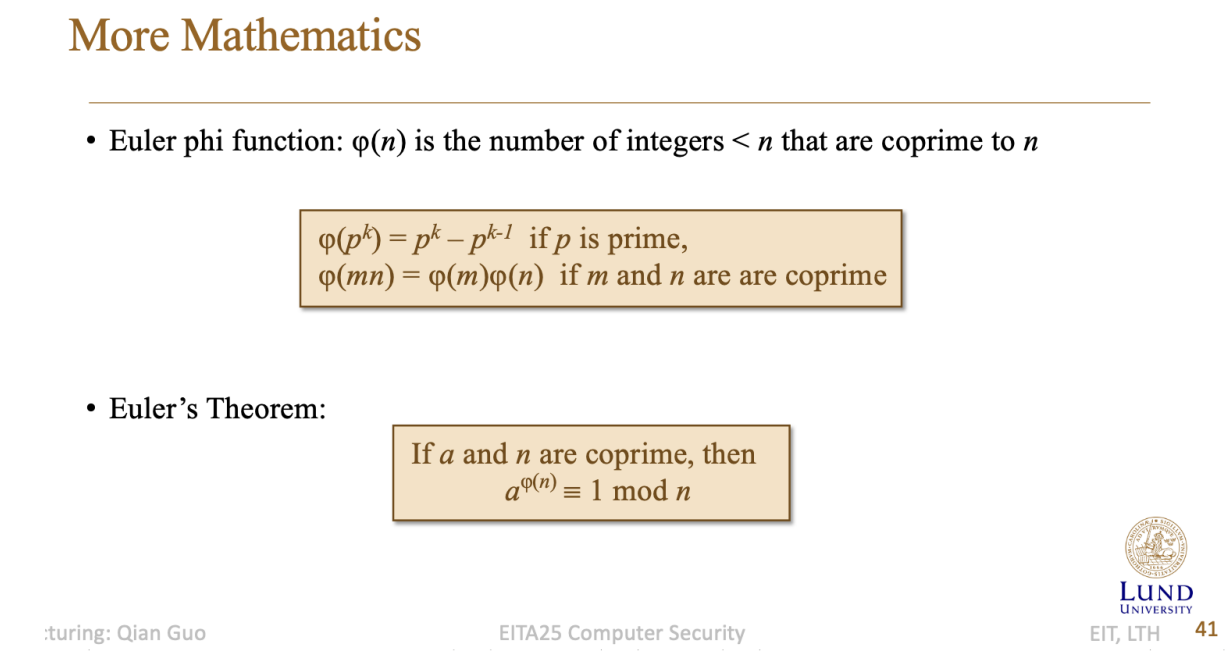
   $(a \times (b+c)) \bmod n = [(a \times b) \bmod n + (a \times c) \bmod n] \bmod n$

   This shows how modular arithmetic interacts with addition and multiplication.

4. **Negative Numbers**: As shown earlier, negative numbers in modular arithmetic are handled by adjusting to ensure the result is always non-negative.

▼ Euler's totient function

Video: Introduction to Euler's Totient Function!



In number theory, two integers a and b are coprime, relatively prime or mutually prime **if the only positive integer that is a divisor of both of them is 1**. Consequently, any prime number that divides a does not divide b, and vice versa, ie the greatest common deviser (GCD) is 1.

Description of the Algorithm

RSA is a block cipher in which the plaintext and ciphertext are integers between 0 and n - 1 for some n.

Encryption and decryption are of the following form, for some plaintext block M and ciphertext block C:

$$C = M^e \bmod n$$
$$M = C^d \bmod n = (M^e)^d \bmod n = M^{ed} \bmod n$$

Sender and receiver need to know the value of n and e, only the receiver knows the value of d. This is a public-private key encryption algorithm:

- public key = PU = {e,n}
- private key = PR = {d,n}

The following require ments are for the public key encryption:

1. It is possible to find values of *e*, *d*, *n* such that *M^(ed)* mod *n* = *M* for all *M* < *n*.
2. It is relatively easy to calculate *M^e* and *C^d* for all values of *M* < *n*.
3. It is infeasible to determine *d* given *e* and *n*.

Requirement 1-2 are easy to meet, requirement 3 can be met with large values on e and n.

For requirement 1, we need to find a relationship such that:

M^(ed) mod *n* = *M*

The preceding relationship holds if *e* and *d* are multiplicative inverses modulo totient(*n*), where totient(*n*) is the Euler totient function of n, where n is a positive number less than n and coprime to n. (Appendix B)

For p and q, both of which are primes, totient(pq) = totient(p)*totient(q) = (p-1)(q-1).

the relationship between e and d can be expressed as:

(e x d)mod totient(n) = 1 ⇔ d mod totient(n) = e^(-1)

e and d are multiplicative inverses mod totient(n). This is true only if d and e are coprime to totient(n), i.e gcd(d, totient(n)) = 1 (same for e).

Key Generation	
Select p, q	p and q both prime, $p \neq q$
Calculate $n = p \times q$	
Calculate $\phi(n) = (p-1)(q-1)$	
Select integer e	$\gcd(\phi(n), e) = 1;\ 1 < e < \phi(n)$
Calculate d	$de \bmod \phi(n) = 1$
Public key	$KU = \{e, n\}$
Private key	$KR = \{d, n\}$

Encryption	
Plaintext:	$M < n$
Ciphertext:	$C = M^e \pmod{n}$

Decryption	
Ciphertext:	C
Plaintext:	$M = C^d \pmod{n}$

Figure 21.7 **The RSA Algorithm**

Algoritm:

1. Select two prime numbers, *p* = 17 and *q* = 11.
2. Calculate *n* = *p* * *q* = 17 ∗ 11 = 187.
3. Calculate totient(*n*) = (*p* - 1)(*q* - 1) = 16 ∗ 10 = 160.
4. Select *e* such that *e* is coprime to totient(*n*) = 160 and less than totient(*n*); we choose

   e = 7.
5. Determine *d* such that *d***e* mod 160 = 1 and *d* < 160. The correct value is d = 23, because 23 ∗ 7 = 161 = (1 ∗ 160) + 1
6. Now the encryption/decryption can be applied:

Encryption and decryption are of the following form, for some plaintext block M and ciphertext block C:

$$C = M^e \bmod n$$
$$M = C^d \bmod n = (M^e)^d \bmod n = M^{ed} \bmod n$$

The Security of RSA

The Factoring Problem

Three approaches to attacking RSA mathematically:

- Factor n into its two prime factors. This enables calculation of $\phi(n) = (p - 1) \times (q - 1)$, which, in turn, enables determination of $d \equiv e^{-1} \pmod{\phi(n)}$.
- Determine $\phi(n)$ directly, without first determining p and q. Again, this enables determination of $d \equiv e^{-1} \pmod{\phi(n)}$.
- Determine d directly, without first determining $\phi(n)$.

Most RSA cryptanalysis focuses on factorizing n, since this is the hardest step. With current algorithms, determining d appears to be as computationally expensive as factoring n. Factoring large numbers is a tough problem, but improvements in factoring algorithms like the **generalized number field sieve (GNFS)** have made it easier to break smaller RSA keys. For example, RSA-130 was factored more efficiently than RSA-129 thanks to GNFS.

The threat to RSA comes from two factors: increased computing power and better factoring algorithms. A specialized algorithm, the **special number field sieve (SNFS)**, can factor certain numbers faster than GNFS, and future breakthroughs may make factoring even easier. This means that for security, RSA keys should be at least **1024 to 2048 bits** in size.

To further protect RSA from easier factorization, key constraints are recommended:

1. **p** and **q** should have similar lengths (in terms of digits).
2. **p - 1** and **q - 1** should have large prime factors.
3. **gcd(p - 1, q - 1)** should be small.

Additionally, if the public exponent e is less than n and the private exponent d is less than n^(1/4), d can be easily computed.

Timing Attacks

first demonstrated by Paul Kocher, exploit variations in the time it takes to perform cryptographic operations, such as modular exponentiation, depending on the bits of the private key. The attack works by observing how long it takes a system to decrypt ciphertext. If the system's decryption process takes longer for certain bits of the private key, an attacker can use this timing variation to gradually deduce the full private key, bit by bit.

For example, in RSA, modular exponentiation is performed bit by bit, and if a bit is set to 1, the operation takes longer due to additional modular multiplications. By tracking the time it takes to execute certain steps, the attacker can infer whether a bit is 1 or 0. Even though timing differences are generally subtle, they can be sufficient for an attacker to gather enough information to break the system.

Countermeasures:

Several methods can defend against timing attacks:

1. **Constant exponentiation time**: Make all exponentiations take the same time, regardless of the input, though this degrades performance.
2. **Random delays**: Introduce random delays to confuse attackers, but if not done sufficiently, attackers can still compensate for the randomness.
3. **Blinding**: This involves multiplying the ciphertext by a random number before performing the decryption, making it difficult for attackers to track the bits of the private key being processed. RSA Data Security implements this by generating a secret random number, modifying the ciphertext, and applying regular RSA decryption, then adjusting the result with the inverse of the random number. This method incurs a small performance penalty (2-10%).

Overall, while timing attacks are a serious threat, these countermeasures can help secure RSA against them.