ICOM 6045
Public Key Cryptography and PKCS

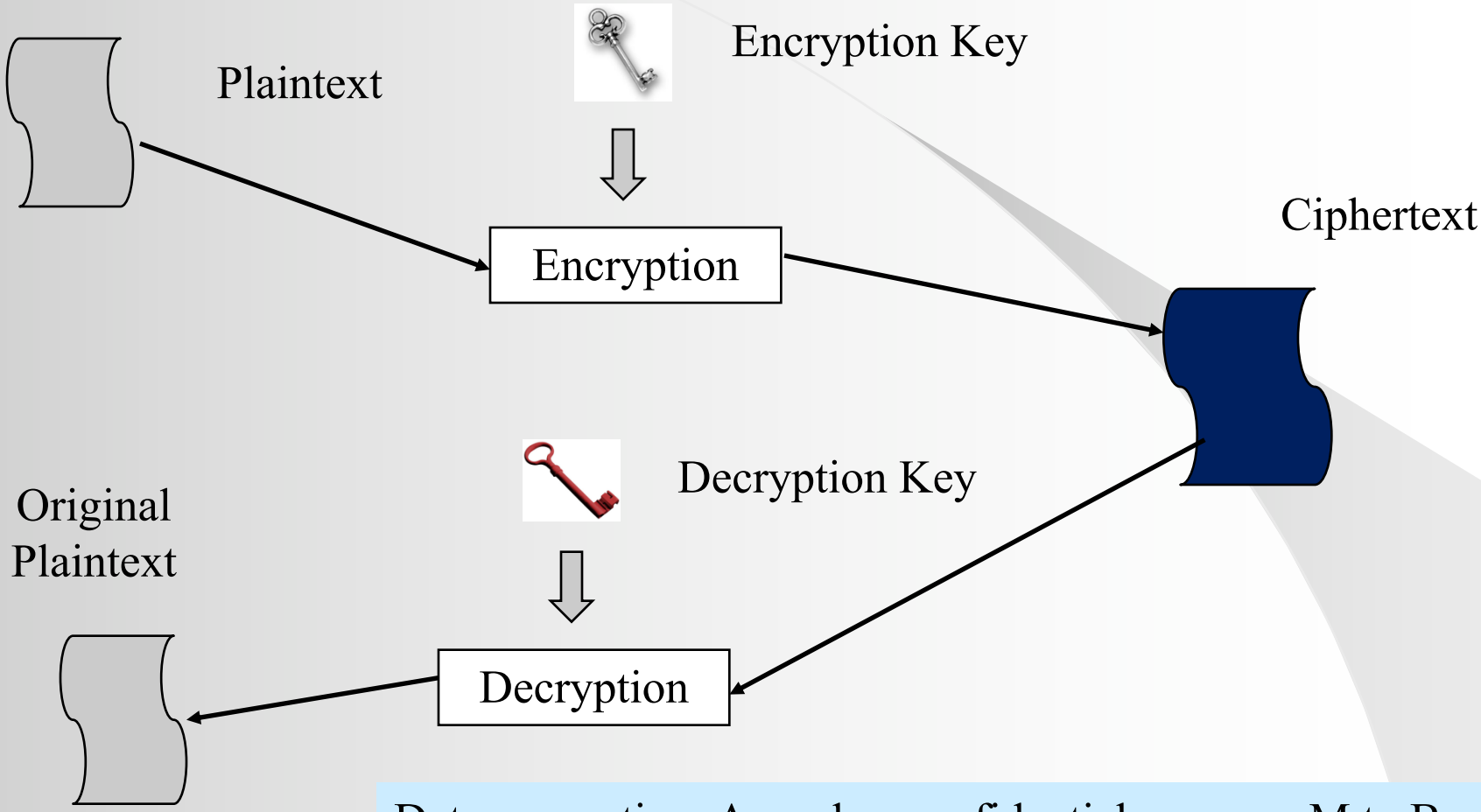
K.P. Chow

University of Hong Kong

Public Key System

- Asymmetric key system or two keys system
- Each party X has 2 keys: public key X_{pub} and private key X_{pri}
- Private key is secret to the owner, public key is open to the public:
  - Mathematically, it is extremely difficult to find the private key given the public key
  - Security strength depends on key length
- Used in *encryption* and *digital signature*
- $D(E(M,X_{pub}),X_{pri}) = M$

Using Public Key System for Encryption



Data encryption: A sends a confidential message M to B

A sends $C=E(M,B_{pub})$ to B

B decrypts by B_{pri}: $M = D(C,B_{pri})$

E is the encryption function and D is the decryption function

What is Digital Signature?

- An authentication tool that can be used
  - To verify the origin of a message (not tampered) and
  - To verify the identity of the sender (non-repudiation), and
  - To resolve any authentication issues between the sender and the receiver
- It should be unforgeable and can be used as a valid signature

Using Public Key System for Digital Signature

- Digital signature: A sends a message M with signature S to B
  - A sends { M + S=Sign(M,A_{pri}) } to B
  - B verifies by checking if Verify(S,A_{pub}) == M
- What are the definitions of Sign & Verify?
  - Sign can be viewed as ***encrypt*** with A's private key
  - Verify can be viewed as ***decrypt*** with A's public key
- Practical usage:
  - Encryption and signature can be used together
  - Combined with hash functions and symmetric key system
  - Use standard formats: PKCS (Public Key Cryptography Standards)

Requirement of Public Key Cryptosystem

- Practical public key cryptosystem depends on discovery of a suitable trap-door one-way function f_k
  - $Y = f_k(X)$ computationally easy if k and X are known
  - $X = f_k^{-1}(Y)$ computationally easy if k and Y are known
  - $X = f_k^{-1}(Y)$ computationally infeasible if Y is known, k is unknown
- f_k uses the public key and f_k^{-1} uses private key
- Example trap-door one-way function
  - Exponentiation: $b = a^e \bmod p$
  - Factorization: $n = p * q$

Exponentiation and Discrete Logarithm

- Tap-door one-way function

  $b = a^x \bmod p$

- The inverse problem to exponentiation is that of finding the discrete logarithm of a number modulo p, i.e., given a & b,

  find x where $a^x = b \bmod p$

- Finding exponentiation is relatively easy: find b given a, x, p
- E.g. let a=3, p=101, i.e. $b = 3^x \bmod 101$
  - if x=3, then b=27;
  - if x=6, then b=729 mod 101=22;

Discrete Logarithm

- The inverse problem to exponentiation is that of finding the discrete logarithm of a number modulo p, i.e.

  find x where $a^x = b \bmod p$

- Given a, b, p, find x (finding **discrete logarithm** is a hard problem)

Discrete Logarithm Example

- E.g. let a=3, p=101, i.e. $b = 3^x \bmod 101$
- If b=3, i.e. $3 = 3^x \bmod 101$, what is x?
- If b=27, i.e. $27 = 3^x \bmod 101$, what is x?
- If b=2, i.e. $2 = 3^x \bmod 101$, what is x?

Multiplication and Factorization

- Trap-door one-way function f_k is *multiplication* and f_k^{-1} is *factorization*
- Multiplication: given 2 number, compute their product is easy
  - E.g. p is 47 and q is 61, then p*q = 2867
- Factorization: given an integer n, find its prime factors
- E.g. what are the prime factors of 2773?

RSA Cryptosystem

- Invented by R. Riverst, A. Shamir and L. Adleman in 1978
- Used both in encryption and digital signature
- Based on the trap-door one-way function f_k is *multiplication* and f_k^{-1} is *factorization*
- Public key is (e,n) and private key is (d,n)
- To encrypt a message m (<n)
  - compute $c = m^e \bmod n$
- To decrypt a ciphertext c (<n)
  - compute $m = c^d \bmod n$

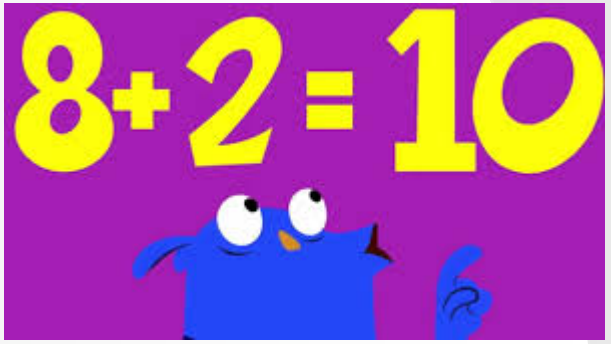
RSA Example

- Public key is (7,143), private key is (103,143)
- Message m is 5
- To encrypt 5, we compute
  - $c = m^e \bmod n = 5^7 \bmod 143 = 47$
- To decrypt 47, we compute
  - $m = c^d \bmod n = 47^{103} \bmod 143 = 5$

Digital Signature using RSA

- Public key is (e,n) and private key is (d,n)
- How about digital signature?
  - Use private key (d,n) to sign a message and public key (e,n) to verify the signature
- To sign a message m (<n)
  - compute $s = m^d \bmod n$
- To verify the signature s of a message m (<n)
  - compute $m1 = s^e \bmod n$
  - check if m1 == m?

Why $M = M^{ed} \bmod n$?



2 Exponential Identities

- $X^a * X^a = X^{(a+b)}$
- E.g. $3^2 * 3^3 = 3^{(2+3)} = 3^5$

- $(X^a)^b = X^{(a*b)}$
- E.g. $(3^2)^3 = 3^{(2*3)} = 3^6$

2 Theorems

- Fermat's theorem: let p be a prime and gcd(m,p)=1, then
  - **$m^{p-1} \bmod p = 1$**
- E.g.
  - $3^{4} \bmod 5 = 81 \bmod 5 = 1$
  - $7^{10} \bmod 11 = 1$

- Euler's Generalization: n = p * q
  - **$m^{(p-1)(q-1)} \bmod n = 1$**
  - p=3, q=5, n=15,
    $4^{(3-1)(4-1)} \bmod 15 = 4^{(2)(3)} \bmod 15 = 4^{(6)} \bmod 15$
    $= 4096 \bmod 15 = 273*15+1 \bmod 15 = 1$
  - p=11, q=5, n=55, $38^{(11-1)(5-1)} \bmod 55 = 1$
    $38^{40} \bmod 55 = 1$

φ(n) [phi of n]

- $\mathbf{m * m^{(p-1)*(q-1)} \bmod n = m * 1}$
- $\varphi(n) = (p-1)*(q-1)$

- $\mathbf{m^{1+(p-1)*(q-1)} \bmod n = m \text{ if } n = p*q}$

Property of encryption and decryption
Let encryption key be e and decryption key be d,

$$e * d = (p-1)*(q-1)+1$$

Encryption: $m^{e} \bmod n \rightarrow c$
Decryption: $c^{d} \bmod n \rightarrow m$

How to find e and d?

An Example

- p = 5, q = 11, n=55
- (p-1) * (q-1) + 1 = 41
- Find e and d such that e*d = 41
- But 41 is a prime? What can we do?

Use modulus with n=40, i.e.

$$e * d = 41 \bmod 40$$

$$e * d = 1 \bmod 40$$

Set e = 3

$$3 * d = 1 \bmod 40$$

Use Extended Euclidean Algorithm, we find d=27

$$3 * 27 = 81 = 1 \bmod 40$$

“Trick” of RSA – uses 2 moduli

- Use one modulus to encrypt:
  $n = p * q$
- Use another modulus to generate the keys:
  $\varphi(n) = (p-1)*(q-1)$
-
- $p = 5$, $q = 11$, $n=55$, $\varphi(n) = 40$
- $e = 3$, $d = 27$
- Encryption: $19^{3} \bmod 55 = 39$
- Decryption: $39^{27} \bmod 55 = 19$

RSA Key Generation

- Generate 2 large primes p, q
- Compute n (the modulus) = p * q
- Compute $\varphi(n) = (p-1)*(q-1)$
- Generate e relatively prime to $(p-1)*(q-1)$, i.e. $\gcd(\varphi(n),e) = 1$
- Compute inverse of e: $d = e^{-1} \bmod ((p-1)*(q-1))$
- Public key is (e,n) and private key is (d,n)
- 1024-bit RSA means n has 1024 bits

RSA Key Generation Example

- Pick p=11, q=13
- Compute n (the modulus) = p * q = 11*13 = 143
- Compute $\varphi(n) = (p-1)*(q-1) = 10*12 = 120$
- Generate e relatively prime to $(p-1)*(q-1)$, i.e. $\gcd(\varphi(n),e) = 1$
  - Select e = 7
- Compute inverse of e: $d = e^{-1} \bmod ((p-1)*(q-1))$
  - $d = 7^{-1} \bmod ((11-1)*(12-1)) = 7^{-1} \bmod (120)$
  - i.e. $7d = 1 \bmod (120)$
  - d = 103
- Public key is (7,143) and private key is (103,143)
- Given the public key (7,143), it is difficult to find 103

How to break RSA?

- Given the public key (e,n), find the private key (d,n)
- Use factorization
- How factorization is related to RSA?

Implementing RSA

- Both encryption and decryption use exponentiation:
  - $c = m^e \bmod n$
  - Exponentiation is repeated multiplication, which takes O(n) multiplication, can it be faster?
  - Use Square & Multiply Exponentiation
- In key generation, we need to compute inverse of e:
  - $d = e^{-1} \bmod ((p-1)*(q-1))$
  - Any efficient algorithm to compute inverse in modulo arithmetic?
  - Use Extended Euclid's Algorithm
- In key generation, we need to generate 2 large primes p, q
  - Any efficient algorithm to generate large prime number?
  - Use probabilistic algorithm

Square & Multiply Exponentiation

- $b = m^e \bmod p$
- Let $e = e_k\, e_{k-1}\, e_{k-2} \ldots e_1$ (e_k is the most significant bit, e_1 is the least significant bit, e is in binary form)

```
d = 1
for j = k downto 1 do {
    d = d * d mod p
    if e_j == 1 then {d = d * m mod p}
}
return d
```

j	d	d'
	1	
5	1	m
4	m^{10}	m^{10}
3	m^{100}	m^{101}
2	m^{1010}	m^{1010}
1	m^{10100}	m^{10100}

- E.g. computer $m^{10100} \pmod p$, then d = 1, 1, m^1, m^{10}, m^{100}, m^{101}, m^{1010}, m^{10100}.
- Total number of multiplications: 7

Computing Inverse in Modulo Arithmetic

- 'Primitive' method : try and error
  - E.g. Inverse of 20 mod 33, i.e. 20 * inv = 1 + k * 33 for some k
  - Try : 33*1 + 1 , 33* 2 + 1 , 33* 3 + 1, etc
  - We get : 34, 67, 100, 133, ….
  - We knows that 20 * 5 = 100 = 1 mod 33
  - So 20^{-1} mod 33 = 5
  - Only work for small numbers
- Extended Euclid's Algorithm
  - The General Solution

Euclid's Algorithm

- To find GCD (Greatest Common Divisor)
- Divide C_0 by C_1, let Quotient = Q_2, Rem = C_2
- E.g. find the GCD of 1769 and 550

C_0	C_1	Quotient	Rem
1769	550	3	119
550	119	4	74

Extended Euclid's Algorithm (1)

- To find multiplicative inverse: $d^{-1} \bmod f$
- Initialize:
  - $A_0 = 1, B_0 = 0, C_0 = f$
  - $A_1 = 0, B_1 = 1, C_1 = d$
- Compute:
  - Quotient $Q_{i+1} = C_{i-1} / C_i$
  - C_{i+1} = Remainder (C_{i-1} / C_i)
  - $B_{i+1} = B_{i-1} - (B_i)\ (Q_{i+1})$
  - $A_{i+1} = A_{i-1} - (A_i)\ (Q_{i+1})$
- Until C_i is 1, d^{-1} (mod) f is B_i
- Find the inverse of 550 mod 1769

i	Q	A_i	B_i	C_i
0	---	1	0	1769 (f)
1	---	0	1	550 (d)
2	3	1	-3	119

Extended Euclid's Algorithm (2)

- For the initial values:
  - $A_0 = 1, B_0 = 0, C_0 = f$
  - $A_1 = 0, B_1 = 1, C_1 = d$
- The following equation is satisfied:
  - $A_0\ f + B_0\ d = C_0$
  - $A_1\ f + B_1\ d = C_1$
- For the following equation:
  - Quotient $Q_{i+1} = C_{i-1} / C_i$
  - C_{i+1} = Remainder (C_{i-1} / C_i)
- We have:
  - $C_{i-1} = C_i\ Q_{i+1} + C_{i+1}$
  - $C_{i+1} = C_{i-1} - C_i\ Q_{i+1}$
- Given
  - $B_{i+1} = B_{i-1} - (B_i)\ (Q_{i+1})$
  - $A_{i+1} = A_{i-1} - (A_1)\ (Q_{i+1})$

Large Primes

- In key generation, we need to generate large primes (prime number that are many hundreds of digits)
- Generate and test method:
  - Take a random number
  - Check if it is prime
- How easy is it to obtain a prime number by random selection?
  - A number that is 2000 bits long, i.e. >= 2^{1999} and <= 2^{2000}, there is about 1 prime in every 1386 numbers
- How to check if a number is prime or not? Naïve approach (very slow):

```
for i = 2 to √n do
    if i is a divisor of n then return prime
return not-prime
```

Primality Testing

- Probabilistic approach: repeatedly running the same test and reduce the probability of error to an acceptable level
- Miller-Rabin Test(n)
  1. Find $k > 0$, q odd so that $n - 1 = 2^k q$
  2. Select a random integer a, $1 < a < n - 1$
  3. if $a^q \bmod n = 1$ then return **inconclusive**
  4. for $j = 0$ to $k - 1$ do
  5. $\quad t = a^j q$
  6. $\quad$ if $a^t \bmod n = n - 1$ then return **inconclusive**
  7. endfor
  8. return **composite**
- If the Miller-Rabin Test(n) returns **inconclusive** t times in succession, then the probability that n is prime $> (1 - 4^{-t})$
  - E.g. if $t = 10$, the probability that n is prime > 0.999999

Communicating using Public Key Cryptography

- Can we use public key cryptography to encrypt a message for communication?
  - YES, but …

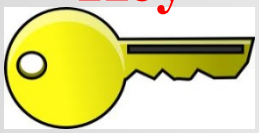
- Public key algorithms are too slow to be used for bulk data encryption
- How can we encrypt large volume of data during communication?

Key Exchange using Public Key Cryptography between A & B

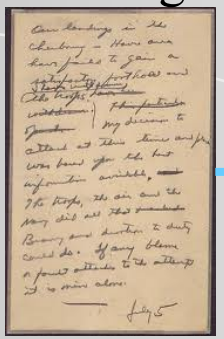
1. A generates secret key (*sk*) randomly
2. A uses B's public key to encrypt *sk* and sends the encrypted key to B
3. B decrypts the encrypted *sk* using his private key
4. Both A and B has the secret key and data can then be encrypted using any block ciphers with *sk*

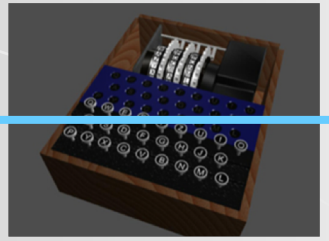
Using symmetric key to encrypt data

Key



Original Message

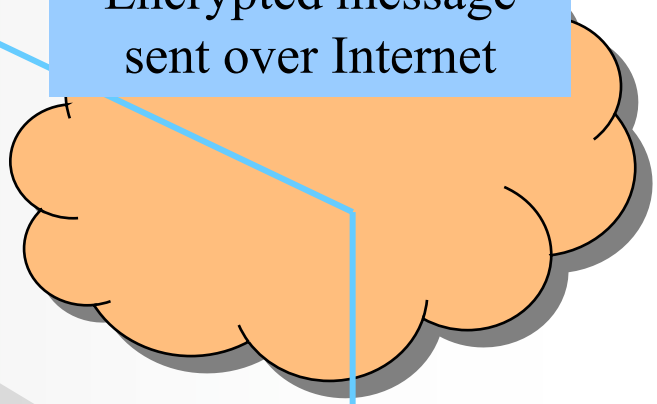




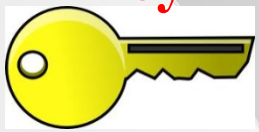
Encryption Algorithm

Encrypted Message

Encrypted message sent over Internet

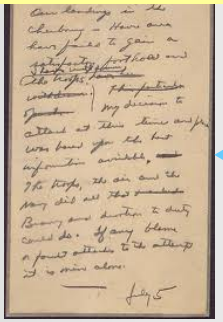


Key

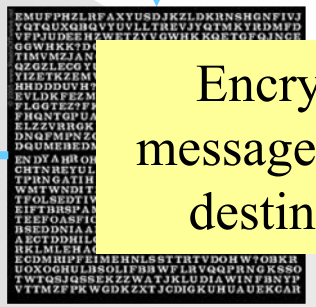


Decryption Algorithm

Original Message

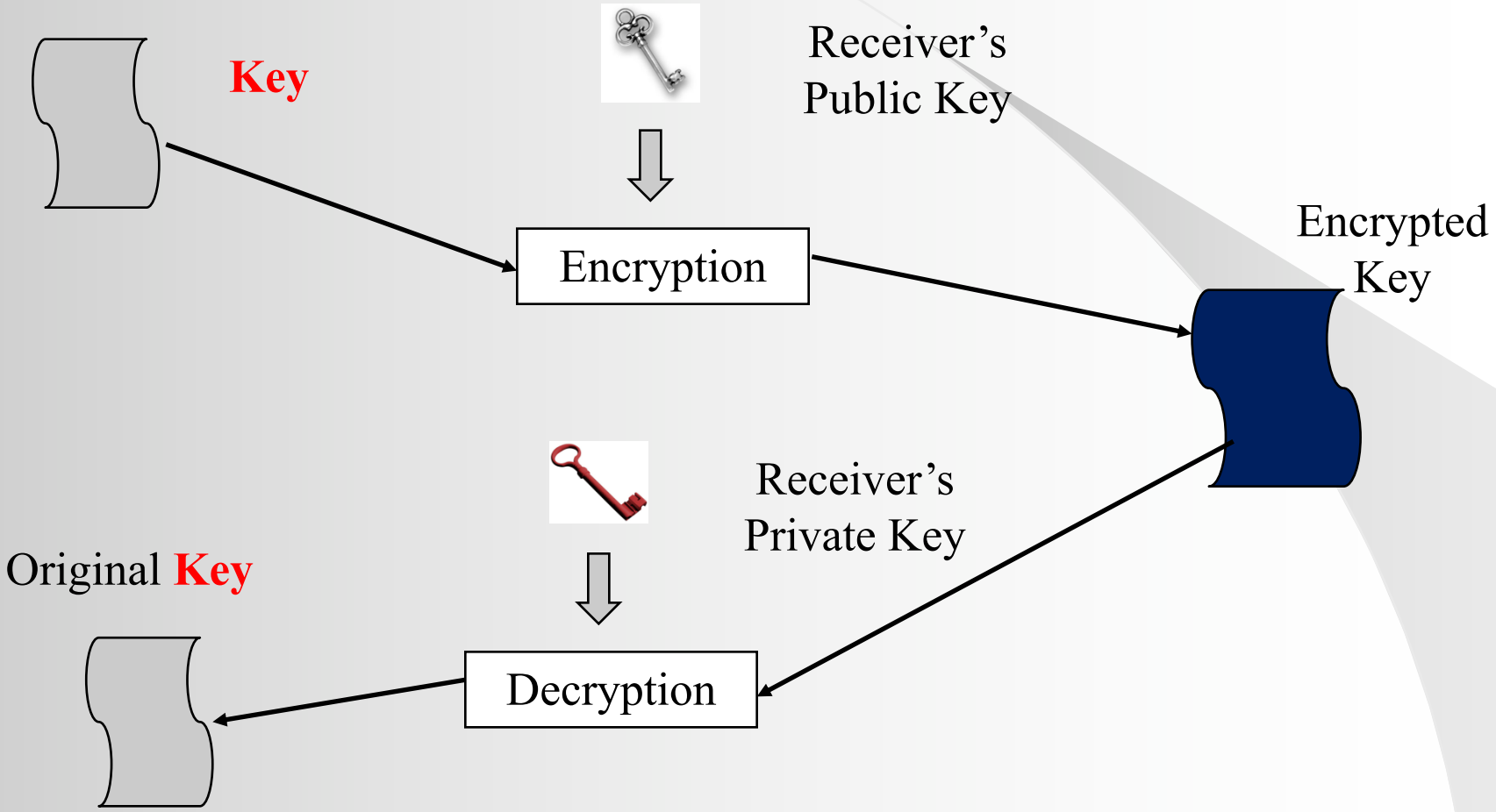


How to distribute the Key?



Encrypted message arrives destination

Using Public Key System for Key Distribution



Key Exchange

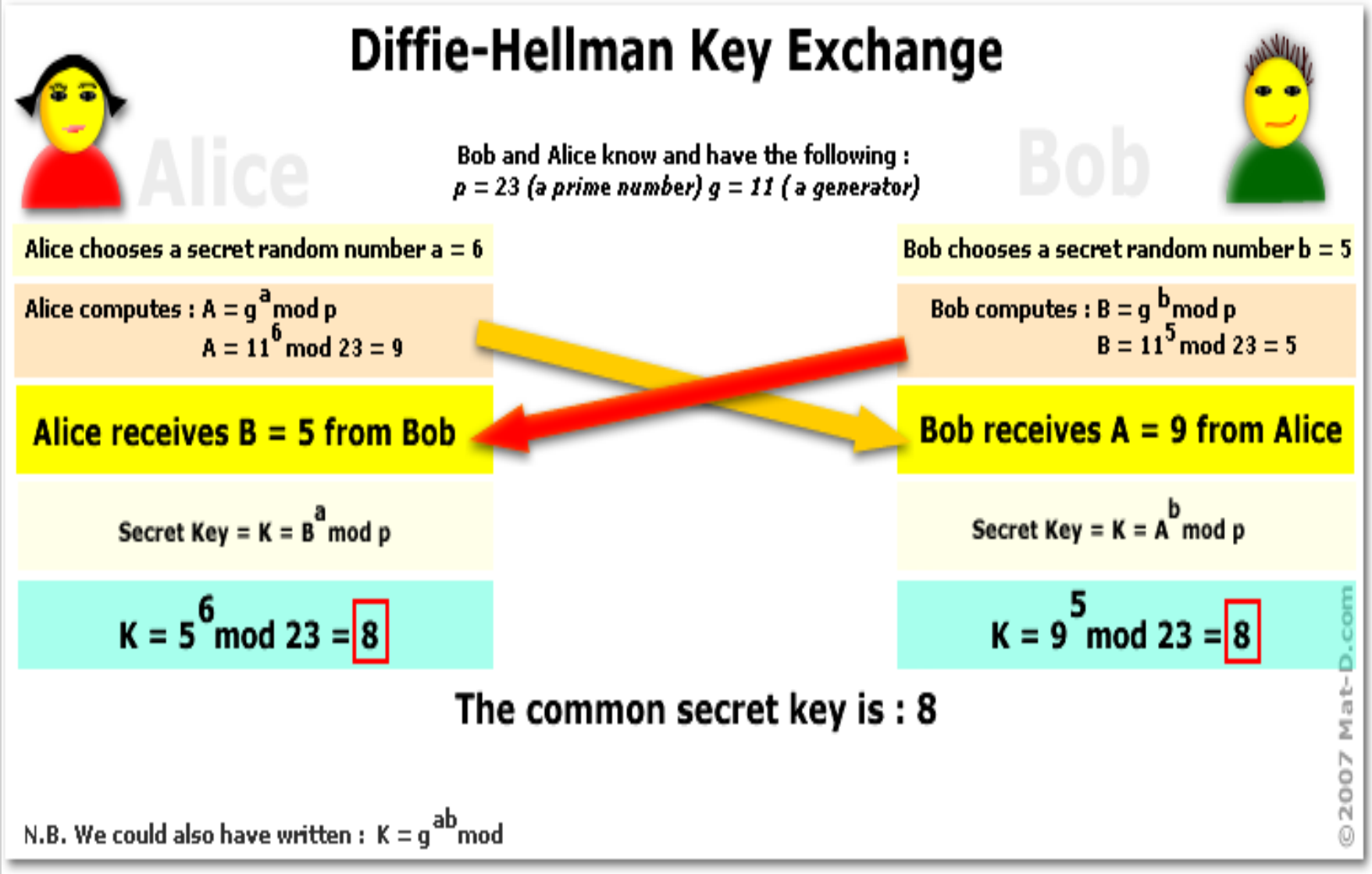
- Using public key algorithms, A generates the secret key and sends it to B
- B has no control on the secret key
- If B wants to participate in key generation, what can we do?
  - Other key exchange algorithm

Diffie-Hellman Key Exchange

Alice				Bob		
Secret	**Public**	**Calculates**	**Sends**	**Calculates**	**Public**	**Secret**
a	p, g				p, g	**b**

Shared secret $s = (g^b)^a \bmod p = (g^a)^b \bmod p$

Diffie-Hellman Key Exchange (Example)



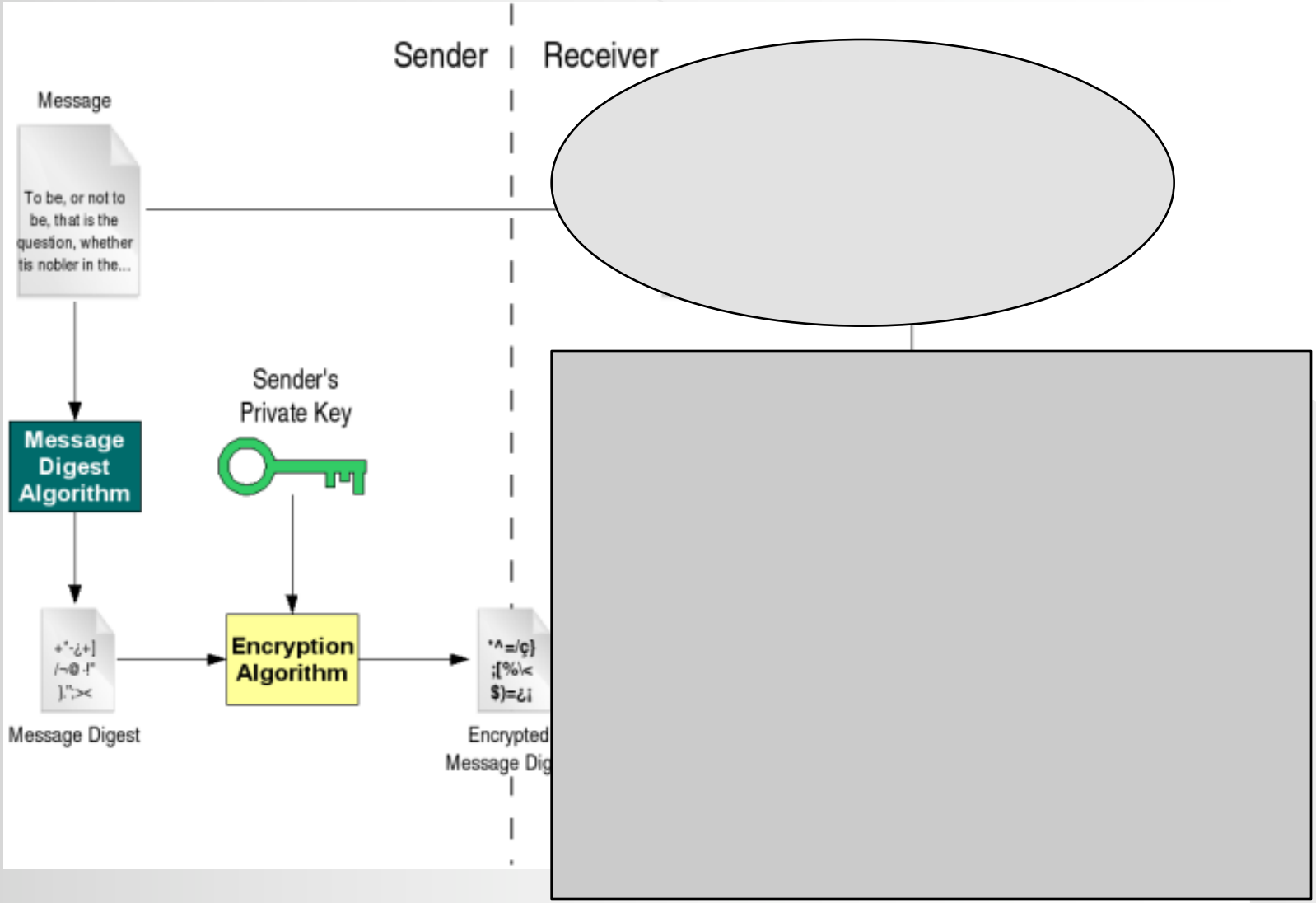
How about digital signature?

- We have a 100MB file, how can we generate it's digital signature using my private key?
  - Are we going to perform exponentiation on the 100MB file
  - It's too slow

Digital Signature

- Similar to encryption, using hashing instead of symmetric key encryption
- Using hashing together with public key cryptography
  - Apply hashing to compute the message digest of the 100MB file
  - Apply public key encryption to sign the message digest, i.e. sign the message digest with signer's private key
  - Signature verification using public key!!!

Digital Signature with Hashing



Communication needs standard!!!

PKCS

- Group of Public Key Cryptography Standards, published by RSA Lab
- Include both algorithm-specific and algorithm-independent implementation standards
- Define an algorithm-independent syntax for digital signatures, digital envelopes (for encryption) and extended certificates
- Two algorithms (RSA and Diffle-Hellman key exchange) are specified in details
- Enable cryptographic algorithm implementers to conform to a standard syntax and achieve interoperability

PKCS (1)

- PKCS #1: RSA Cryptography Standard
  - Fundamental building block for all public key cryptography
  - Defines a method for encrypting data using RSA
  - Includes RSA key generation, key syntax, the encryption and decryption processes, signature algorithms
- PKCS #5: Password-based Cryptography Standard
  - Defines how a password and a random number (salt) are to be mixed to together to form a symmetric key

PKCS (2)

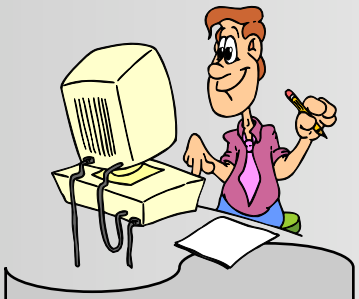
- PKCS #7: Cryptographic Message Syntax Standard
  - Specifies how to package information like signature algorithm, signer's certificate, original message, signature bytes, … in a standard format
  - Allows senders to encode the objects and receivers to decode the information in a predictable, interoperable way, using independent implementation
  - Includes both signed data and enveloped data
- PKCS #8: Private-key Information Syntax Standard
  - Provides standard definitions for encoding and decoding private keys either in a raw form or in an encrypted format

PKCS (3)

- PKCS #10: Certification Request Syntax Standard
- PKCS #12: Personal Information Exchange Syntax Standard
  - A transfer syntax for personal identity information, including private keys, certificates, …

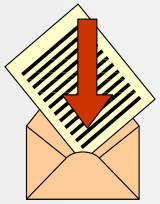
Signed Message with PKCS #7 and S/MIME (1)

Bob sends a signed message to Alice.



Dear Alice,
How are you? …

Bob's Message



2

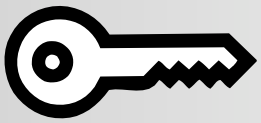
p7s S/MIME Message

MIME header

Message

SignedData

1



Bob's private key

Algorithm: MD5withRSA
SignerInfo: …
Signature: …

Signed Message with PKCS #7 and S/MIME (2)

- Once Bob composes his message and click Send:
  - Bob's private key is extracted from the application's defined certificate database
  - The application then feeds the private key and message contents into a signing algorithm (such as MD5withRSA) and generates a signature
  - The signature together with other information (SignerInfo, …) is bundled into a signature-only p7s SignedData object
  - The application constructs a p7s S/MIME message with MIME header information, original message, and SignedData object
  - The application sends the S/MIME message to Alice

Verifying Signed Message with PKCS #7 and S/MIME (1)

Alice verifies the signed message received from Bob.

p7s S/MIME Message

MIME header

Message

SignedData

Algorithm: MD5withRSA
SignerInfo: …
Signature: …

1

2

3

3

Bob's Certificate

Bob's Message

Dear Alice,
How are you? …

Bob Smith

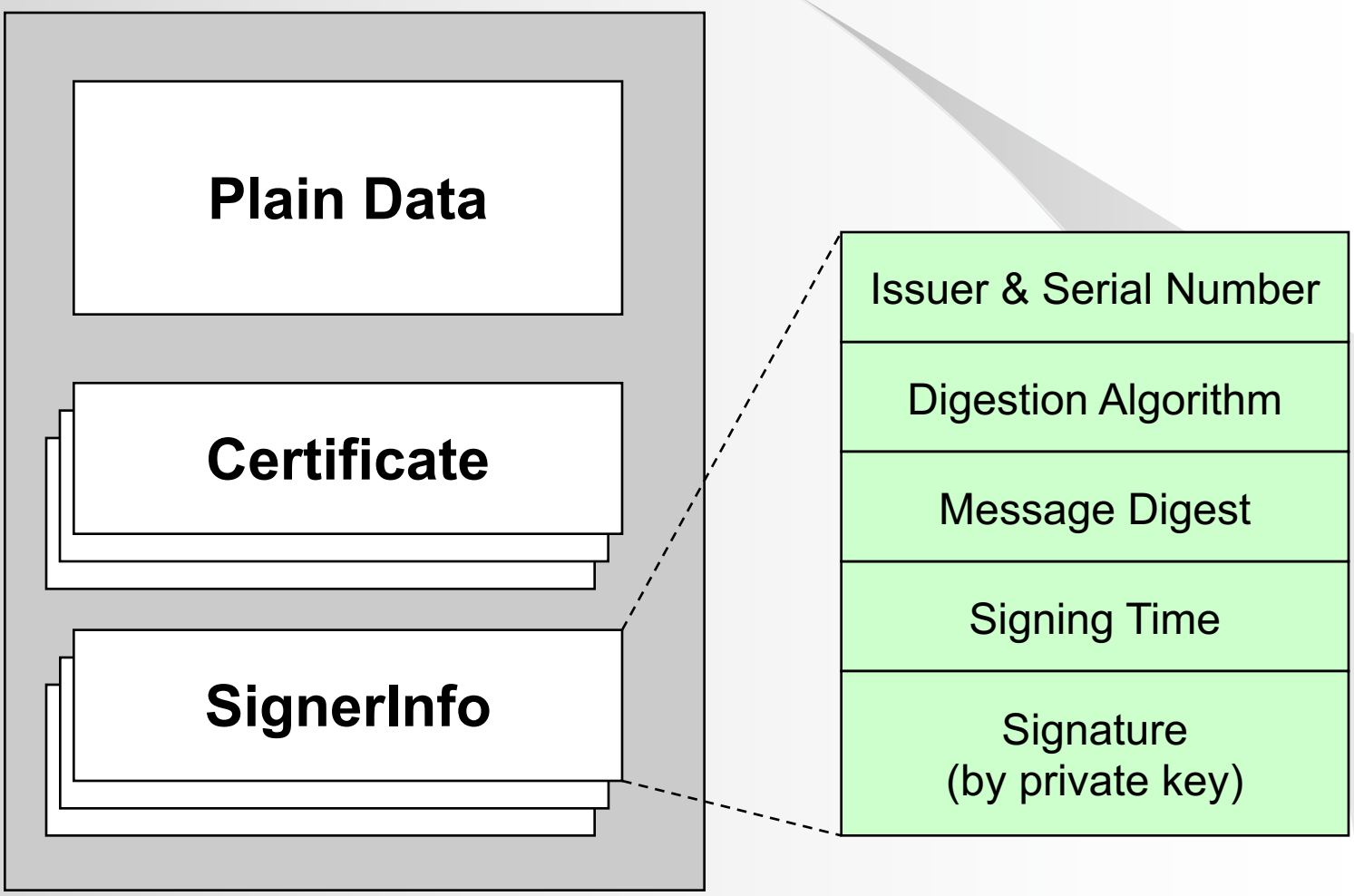
Bob's Signature

Bob's Public Key

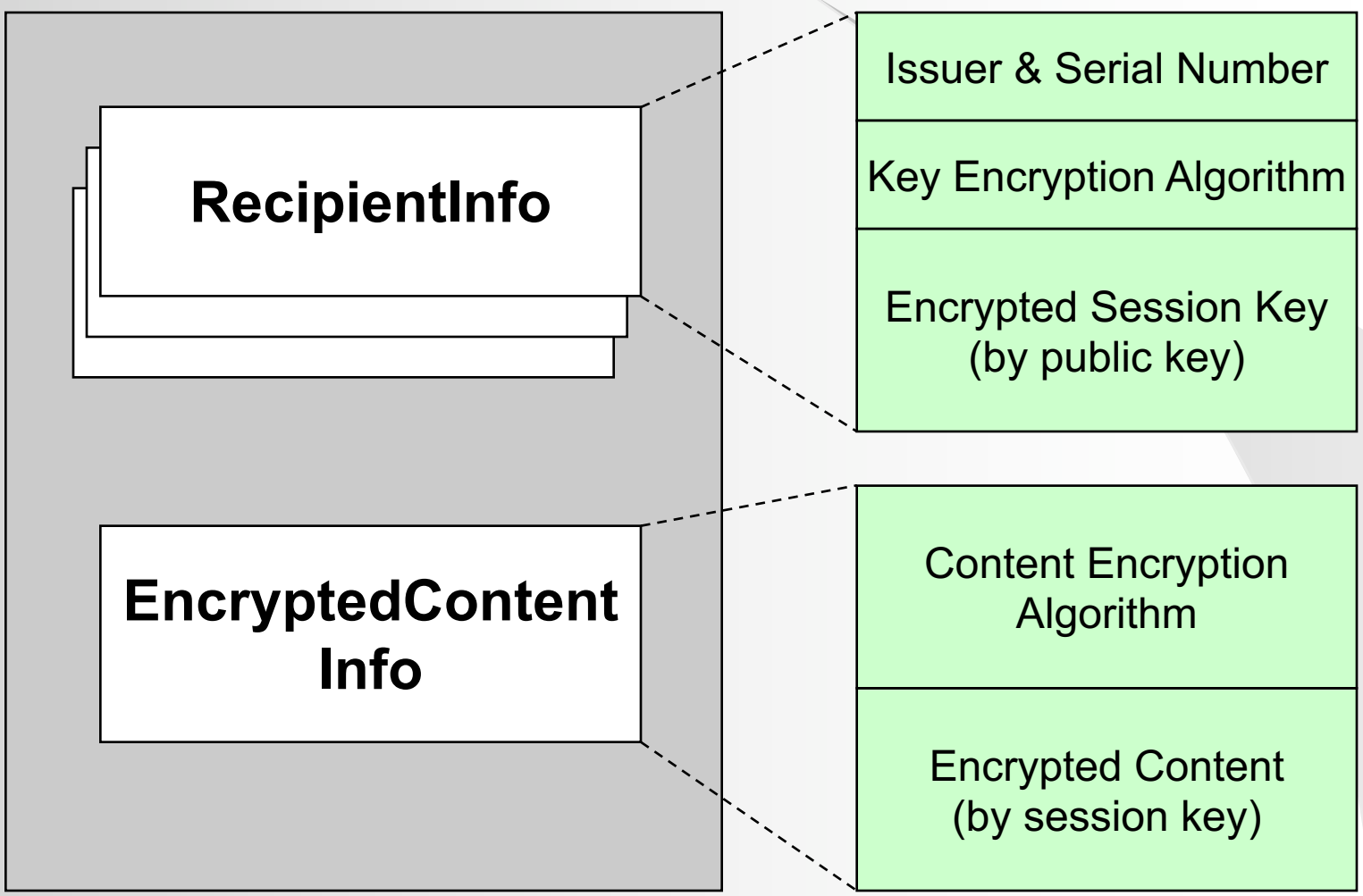
Verifying Signed Message with PKCS #7 and S/MIME (2)

- When Alice's mail client receives Bob's message:
  - The mail client first verifies the MIME headers in the S/MIME message
  - If the header indicates a pkcs7-signature type, the mail client extracts the signer's certificate corresponding to each SignerInfo object from the SignedData object
  - The mail client verifies the signature by tracing the certificate chain back to a known trusted root CA
  - If the certificate is verified, the mail client extracts the signer's public key from the certificate and obtains the signature from each SignerInfo object in the SignedData object
  - The mail client then verifies each signature with the message and the corresponding public key, returns true if the verification is successful, otherwise false

PKCS#7 SignedData Message



PKCS#7 EnvelopedData Message



PKCS#7 SignedAndEnvelopedData Message