# Public Key Cryptography

# Introduction

- Foundation of today’s secure communication
- Allows communicating parties to obtain a shared secret key
- Public key (for encryption) and Private key (for decryption)
- Private key (for digital signature) and Public key (to verify signature)

# Brief History Lesson

- Historically same key was used for encryption and decryption
- Challenge: exchanging the secret key (e.g. face-to-face meeting)
- 1976: Whitfield Diffie and Martin Hellman
  - key exchange protocol
  - proposed a new public-key cryptosystem
- 1978: Ron Rivest, Adi Shamir, and Leonard Adleman (all from MIT)
  - attempted to develop a cryptosystem
  - created RSA algorithm

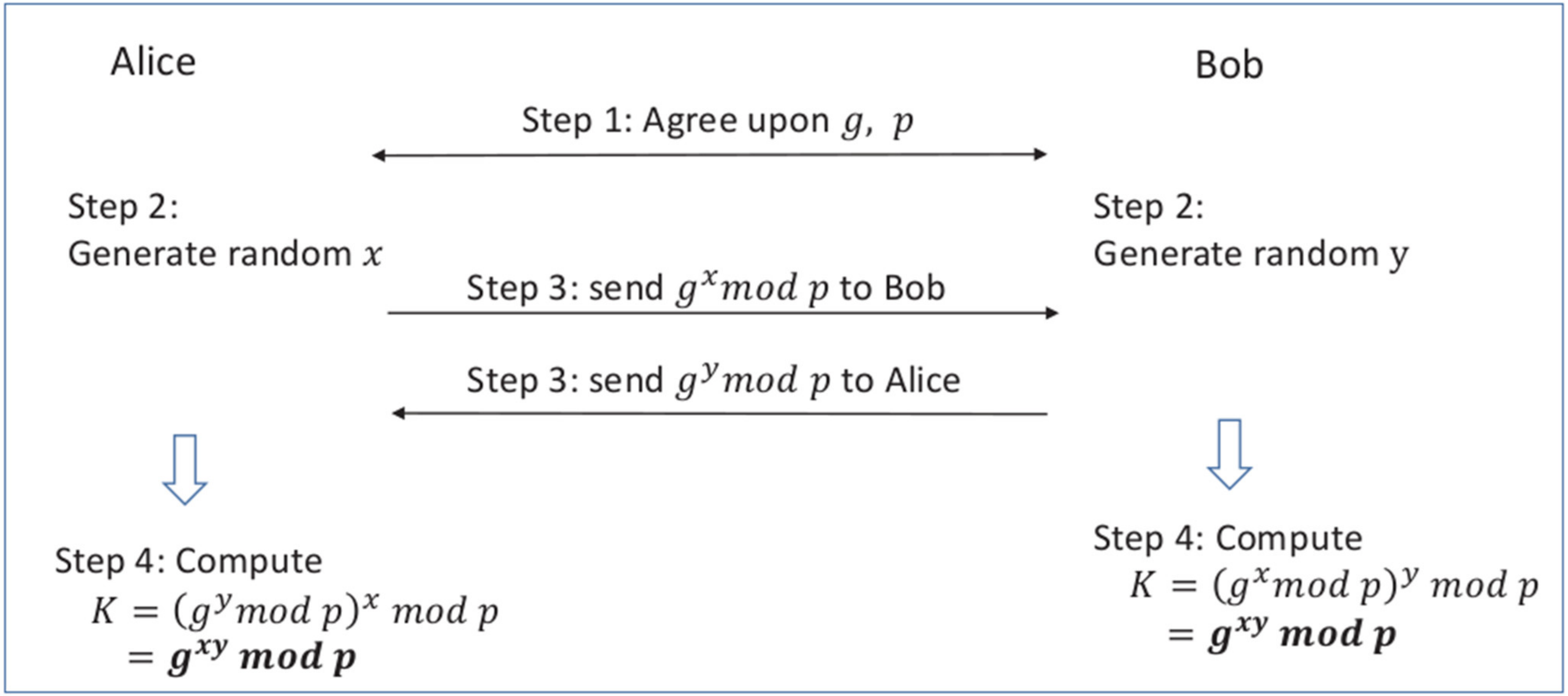
# Outline

- Public-key algorithms
  - Diffie-Hellman key exchange
  - RSA algorithm
  - Digital signature
- Public-key infrastructure
- SSL/TLS protocol

# Diffie-Hellman Key Exchange

- Allows communicating parties with no prior knowledge to exchange shared secret keys over an insecure channel
- Alice and Bob want to communicate
- Alice and Bob agree on:
  - Number p: big prime number (such as a 2048-bit number)
  - Generator g: small prime number (such as 2 and 3)
- Alice picks a random positive integer $x < p$
- Bob picks a random positive integer $y < p$

# Diffie-Hellman Key Exchange (Contd.)



# Turn DH Key Exchange into a Public-Key Encryption Algorithm

- DH key exchange protocol allows exchange of a secret
- Protocol can be tweaked to turn into a public-key encryption scheme
- Need:
  - Public key: known to the public and used for encryption
  - Private key: known only to the owner, and used for decryption
  - Algorithm for encryption and decryption

# Turn DH Key Exchange into a Public-Key Encryption Algorithm (Contd.)

**Alice**

**Bob**

Key Generation

Public key: $\boldsymbol{g},\ \boldsymbol{p},\ \boldsymbol{g^x mod\ p}$

Private key: $\boldsymbol{g},\ \boldsymbol{p},\ \boldsymbol{x}$

Alice's Public key

Encryption

- Generate a random number $y$
- Compute $K = (g^x mod\ p)^y\ mod\ p$ $= g^{xy}\ mod\ p$
- Ciphertext $C = ENCRYPT_K(M)$

$C,\ g^y\ mod\ p$

Decryption

- Compute $K = (g^y mod\ p)^x\ mod\ p$ $= g^{xy}\ mod\ p$
- Plaintext $M = DECRYPT_K(C)$

# RSA Algorithm

We will cover:

- Modulo Operation
- Euler’s Theorem
- Extended Euclidean Algorithm
- RSA Algorithm
- Algorithm example on small and large number

# Modulo Operation

- The RSA algorithm is based on modulo operations
- `a mod n` is the remainder after division of a by the modulus n
- Second number is called modulus
- For example, (10 mod 3) equals to 1 and (15 mod 5) equals to 0
- Modulo operations are distributive:

$$
\begin{aligned}
(a+b) \mod n &= [(a \mod n) + (b \mod n)] \mod n \\
a * b \mod n &= [(a \mod n) * (b \mod n)] \mod n \\
a^x \mod n &= (a \mod n)^x \mod n
\end{aligned}
$$

# Euler's Theorem

- Euler's totient function $\varphi(n)$ counts the positive integers up to a given integer n that are relatively prime to n
- $\varphi(n) = n - 1$, if n is a prime number.
- Euler's totient function property:
  - if m and n are relatively prime, $\varphi(mn) = \varphi(m) * \varphi(n)$
- Euler's theorem states:
  - $a^{\varphi(n)} = 1 \pmod{n}$

# Euler’s Theorem (Contd.)

Example: to calculate $4^{100003} \bmod 33$

- $\varphi(33) = \varphi(3) * \varphi(11) = (3 - 1) * (11 - 1) = 20$
- $100003 = 5000\varphi(33) + 3$

$$
\begin{aligned}
4^{100003} &\mod 33 \\
&= 4^{20*5000+3} \mod 33 \\
&= (4^{20})^{5000} * 4^3 \mod 33 \\
&= [(4^{20})^{5000} \mod 33)] * 4^3 \mod 33 \ \text{(applying distributive rule)} \\
&= [\mathbf{(4^{20} \mod 33)}]^{5000} * 4^3 \mod 33 \ \text{(applying distributive rule)} \\
&= 1^{5000} * 64 \mod 33 \ \text{(applying Euler's theorem)} \\
&= 31.
\end{aligned}
$$

# Extended Euclidean Algorithm

- Euclid's algorithm: efficient method for computing GCD
- Extended Euclidean algorithm:
  - computes GCD of integers a and b
  - finds integers x and y, such that: $ax + by = \gcd(a, b)$
- RSA uses extended Euclidean algorithm:
  - e and n are components of public key
  - Find solution to equation:
    $e * x + \varphi(n) * y = \gcd(e, \varphi(n)) = 1$
  - x is private key (also referred as d)
  - Equation results: $e * d \bmod \varphi(n) = 1$

# RSA Algorithm

We will cover:

- Key generation
- Encryption
- Decryption

# RSA: Key Generation

- Need to generate: modulus n, public key exponent e, private key exponent d
- Approach
  - Choose p,q (large random prime numbers)
  - $n = pq$ (should be large)
  - Choose e, $1 < e < \varphi(n)$ and e is relatively prime to $\phi(n)$
  - Find d, $ed \bmod \varphi(n) = 1$
- Result
  - (e,n) is public key
  - d is private key

# RSA: Encryption and Decryption

- Encryption
  - treat the plaintext as a number
  - assuming M < n
  - `C = M`$^e$ `mod n`
- Decryption
  - `M = C`$^d$ `mod n`

$$
\begin{aligned}
M^{ed} \mod n &= M^{k\phi(n)+1} \mod n \\
&= M^{k\phi(n)} * M \mod n \\
&= (\boldsymbol{M^{\phi(n)} \ \mathbf{mod}\ n})^k * M \mod n \quad \text{(applying distributive rule)} \\
&= 1^k * M \mod n \quad \text{(applying Euler's theorem)} \\
&= M
\end{aligned}
$$

# RSA Exercise: Small Numbers

- Choose two prime numbers p = 13 and q = 17
- Find e:
  - $n = pq = 221$
  - $\varphi(n) = (p - 1)(q - 1) = 192$
  - choose e = 7 (7 is relatively prime to $\phi(n)$)
- Find d:
  - $ed = 1 \bmod \varphi(n)$
- Solving the above equation is equivalent to: $7d + 192y = 1$
- Using extended Euclidean algorithm, we get d = 55 and y = −2

# RSA Exercise: Small Numbers (Contd.)

Encrypt M = 36

$$\boxed{\begin{aligned} M^e \quad \bmod n &= 36^7 \quad \bmod 221 \\ &= (36^2 \quad \bmod 221)^3 * 36 \quad \bmod 221 \\ &= 191^3 * 36 \quad \bmod 221 \\ &= 179 \quad \bmod 221. \end{aligned}}$$

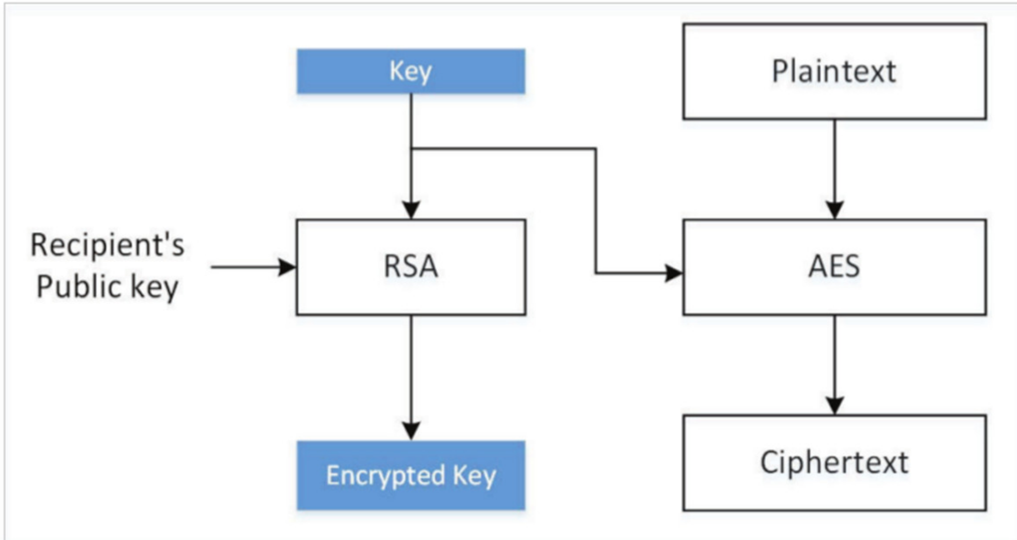
Cipher text ( C ) = 179

# RSA Exercise: Small Numbers (Contd.)

$$
\begin{aligned}
C^d \mod n &= 179^{55} \mod 221 \\
&= (179^2 \mod 221)^{27} * 179 \mod 221 \\
&= 217^{27} * 179 \mod 221 \\
&= (217^2 \mod 221)^{13} * 217 * 179 \mod 221 \\
&= 16^{13} * 217 * 179 \mod 221 \\
&= (16^2 \mod 221)^{6} * 16 * 217 * 179 \mod 221 \\
&= 35^{6} * 16 * 217 * 179 \mod 221 \\
&= (35^2 \mod 221)^{3} * 16 * 217 * 179 \mod 221 \\
&= 120^{3} * 16 * 217 * 179 \mod 221 \\
&= (120^2 \mod 221) * 120 * 16 * 217 * 179 \mod 221 \\
&= 35 * 120 * 16 * 217 * 179 \mod 221 \\
&= 36 \mod 221
\end{aligned}
$$

# Hybrid Encryption

- High computation cost of public-key encryption
- Public key algorithms used to exchange a secret session key
- Key (content-encryption key) used to encrypt data using a symmetric-key algorithm



# Using OpenSSL Tools to Conduct RSA Operations

We will cover:

- Generating RSA keys
- Extracting the public key
- Encryption and Decryption

# OpenSSL Tools: Generating RSA keys

Example: generate a 1024-bit public/private key pair

- `openssl genrsa -aes128 -out private.pem 1024`
- private.pem: Base64 encoding of DER generated binary output

```
$ more private.pem
-----BEGIN RSA PRIVATE KEY-----
MIICWgIBAAKBgQCuXJawrRzJNG9vt2Zqe+/TCT3OxuEKRWkHfE5uZBkLCMgGbYzK
...
mesOrjIfm0ljUNL4VRnrLxrl/1xEBGWedCuCPqeV
-----END RSA PRIVATE KEY-----
```

# OpenSSL Tools: Generating RSA keys (Contd.)

Actual content of private.pem

```
$ openssl rsa -in private.pem -noout -text
Enter pass phrase for private.pem:
Private-Key: (1024 bit)
modulus:
    00:c4:5a:9d:8d:f7:ad:0d:e7:60:4e:b3:9c:76:93: ...
publicExponent: 65537 (0x10001)
privateExponent:
    00:a5:86:fe:6b:3f:f0:53:58:4a:88:0e:42:48:74: ...
prime1:
    00:ec:a0:f7:02:8d:79:a0:8b:c5:5b:e6:a0:25:2c: ...
prime2:
    00:d4:6d:9c:4a:35:6b:fb:db:42:20:d8:6e:45:a9: ...
exponent1:
    06:72:d4:88:73:46:8f:43:7f:db:63:4b:95:f7:c4: ...
exponent2:
    00:d1:3c:45:bd:32:71:72:59:bd:00:ed:2d:70:a0: ...
coefficient:
    22:f5:95:05:81:c4:fd:3e:52:99:16:b5:66:92:52: ...
```

# OpenSSL Tools: Extracting Public Key

- `openssl rsa -in private.pem -pubout > public.pem`
- Content of public.pem:

```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDEWp2N960N52BOs5x2k53WglVn
iAv5oUemZdfnGP1qUhTMZfhSbD27eOUJZAEdrMS/4Nax/BJIxz6N+L2K2cQQasJY
Gqf1PetXKtYakzgd5dBuB3aogOTJaBSt8/A0DBK2MtwNMnBxeZWnf4DK8Glsbp2S
nsGmCdceQ4nelGZbIwIDAQAB
-----END PUBLIC KEY-----
```

```
$ openssl rsa -in public.pem -pubin -text -noout
Public-Key: (1024 bit)
Modulus:
    00:af:1a:d9:ca:91:91:6b:b6:d0:1d:56:7a:1b:2d: ...
Exponent: 65537 (0x10001)
```

# OpenSSL Tools: Encryption and Decryption

- Plain Text

```
$ echo "This is a secret." > msg.txt
```

- Encryption

```
$ openssl rsautl -encrypt -inkey public.pem -pubin \
                 -in msg.txt -out msg.enc
```

- Decryption

```
$ openssl rsautl -decrypt -inkey private.pem -in  msg.enc
Enter pass phrase for private.pem:
This is a secret.
```

# Paddings for RSA

- Secret-key encryption uses encryption modes to encrypt plaintext longer than block size.
- RSA used in hybrid approach (Content key length << RSA key length)
- To encrypt:
  - short plaintext: treat it a number, raise it to the power of e (modulo n)
  - large plaintext: use hybrid approach (treat the content key as a number and raise it to the power of e (modulo n)
- **Treating plaintext as a number and directly applying RSA is called plain RSA or textbook RSA**

# Attacks Against Textbook RSA

- RSA is deterministic encryption algorithm
  - same plaintext encrypted using same public key gives same ciphertext
  - secret-key encryption uses randomized IV to have different ciphertext for same plaintext
- For small e and m
  - if $m^e <$ modulus n
  - e-th root of ciphertext gives plaintext
- If same plaintext is encrypted e times or more using the same e but different n, then it is easy to decrypt the original plaintext message via the Chinese remainder theorem

# Paddings: PKCS#1 v1.5 and OAEP

- Simple fix to defend against previous attacks is to add randomness to the plaintext before encryption
- Approach is called padding
- Types of padding:
    - PKCS#1 (up to version 1.5): weakness discovered since 1998
    - Optimal Asymmetric Encryption Padding (OAEP): prevents attacks on PKCS
- `rsautl` command provides options for both types of paddings (PKCS#1 v1.5 is default)

# PKCS Padding

- Plaintext is padded to 128 bytes
- Original plaintext is placed at the end of the block
- Data inside the block (except the first two bytes) are all random numbers
- First byte of the padding is always 00 (so that padded plaintext as integer is less than modulus n)
- Second byte is 00, 01, and 02 (different strings used for padding for different types)

# PKCS Padding (Contd.)

```
$ openssl rsautl -encrypt -inkey public.pem -pubin \
                 -in msg.txt -out msg.enc -pkcs


$ openssl rsautl -decrypt -inkey private.pem \
                 -in msg.enc -out newmsg.txt -raw

$ xxd newmsg.txt
00000000: 0002 1b19 331a 1ea8 049e 8667 3b55 057c  ....3......g;U.|
00000010: 1072 e2bb 0aca 9af0 dd0e 5706 b34d e4a3  .r........W..M..
00000020: 7df6 b4d3 5f9b 8303 5ce7 67ee 150e 0fe1  }..._...\.g.....
00000030: f73f 6dc4 af36 117d 0d63 72f1 88f2 337f  .?m..6.}.cr...3.
00000040: 100b afac 8b26 fa65 d5a6 10b3 cf10 0b35  .....&.e.......5
00000050: 171b 9cc2 3409 c3b6 d953 a8a4 4617 4356  ....4....S..F.CV
00000060: 3f5f 1a91 9a97 5863 eae2 8ec5 4a00 5468 ?_....Xc....J.Th
00000070: 6973 2069 7320 6120 7365 6372 6574 2e0a  is is a secret..
```

# OAEP Padding

- Original plaintext is not directly copied into the encryption block
- Plaintext is XORed with a value derived from random padding data

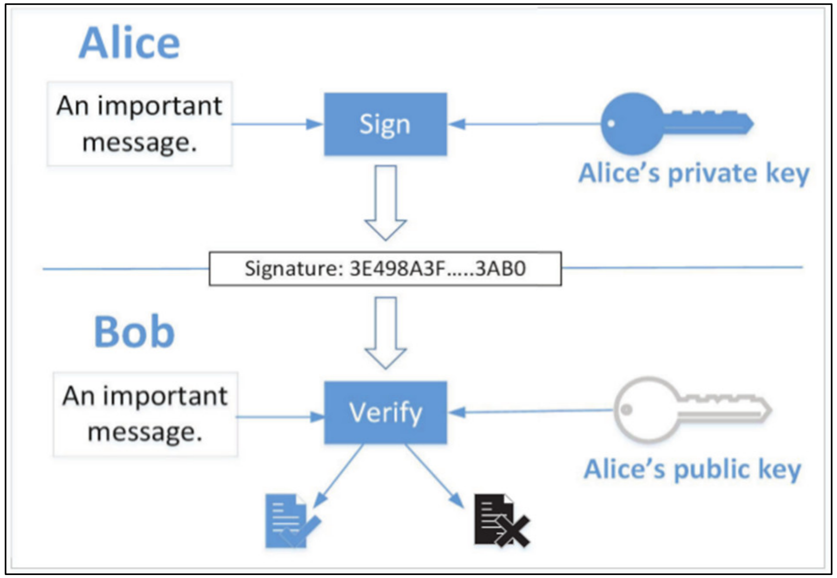
```
$ openssl rsautl -encrypt -inkey public.pem -pubin \
                 -in msg.txt -out msg.enc -oaep

$ openssl rsautl -decrypt -inkey private.pem \
                 -in msg.enc -out newmsg.txt -raw

$ xxd newmsg.txt
00000000: 006f 5f5e 5e0d e813 7fb0 3d45 e1ed d4fa  .o_^^.....=E....
00000010: 0688 1196 bb47 4501 b815 8922 51a0 5184  .....GE...."Q.Q.
00000020: d6b1 9819 4c00 07d1 b985 0248 8822 7b4f  ....L......H."{O
00000030: 8470 b195 1e4e 288f db91 f905 9d70 01de  .p...N(......p..
00000040: e0f4 5b4c 5b8a 26df 7031 b4a6 6547 d07d  ..[L[.&.p1..eG.}
00000050: e8ca 0006 3b65 a3ba 0f9f f865 6e80 6e0d  ....;e.....en.n.
00000060: 04ff 82a1 2c0b 3d1d 8d63 19b1 56f7 14f8  ....,.=..c..V...
00000070: 880e d003 d0e8 003c 9818 b083 7ba0 c6e6  .......<....{...
```

# Digital Signature

- Goal: provide an authenticity proof by signing digital documents
- Diffie-Hellman authors proposed the idea, but no concrete solution
- RSA authors developed the first digital signature algorithm



# Digital Signature using RSA

- Apply private-key operation on m using private key, and get a number s, everybody can get the m back from s using our public key
- For a message m that needs to be signed:

  Digital signature = $m^d \bmod n$

- In practice, message may be long resulting in long signature and more computing time
- Instead, we generate a cryptographic hash value from the original message, and only sign the hash

# Digital Signature using RSA (Contd.)

Generate message hash

```
# Generate the hash from the message
$ openssl sha256 -binary msg.txt > msg.sha256
$ xxd msg.sha256
00000000: 8272 61ce 5ddc 974b 1b36 75a3 ed37 48cd  .ra.]..K.6u..7H.
00000010: 83cd de93 85f0 6aab bd94 f50c db5a b460  ......j......Z.`
```

# Digital Signature using RSA (Contd.)

Generate and verify the signature

```
# Sign the hash
$ openssl rsautl -sign -inkey private.pem -in msg.sha256 -out msg.sig

# Verify the signature
$ openssl rsautl -verify -inkey public.pem -in msg.sig -pubin \
                 -raw | xxd
00000000: 0001 ffff ffff ffff ffff ffff ffff ffff  ................
00000010: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000020: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000030: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000040: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000050: ffff ffff ffff ffff ffff ffff ffff ff00  ................
00000060: 8272 61ce 5ddc 974b 1b36 75a3 ed37 48cd  .ra.]..K.6u..7H.
00000070: 83cd de93 85f0 6aab bd94 f50c db5a b460  ......j......Z.`
```

# Attack Experiment on Digital Signature

- Attackers cannot generate a valid signature from a modified message because they do not know the private key
- If attackers modifies the message, the hash will change and it will not be able to match with the hash produced from the signature verification
- Experiment: modify 1 bit of signature file msg.sig and verify the signature

# Attack Experiment on Digital Signature (Contd.)

After applying the RSA public key on the signature, we get a block of data that is significantly different

```
$ openssl rsautl -verify -inkey public.pem -in msg.sig -pubin \
                 -raw | xxd
00000000: 8116 cdc6 6b45 bcfc 98c3 7b09 514e 82fd  ....kE....{.QN..
00000010: 88a2 170b 414d 1ce8 7d18 d031 f03e db9f  ....AM..}..1.>..
00000020: 6f0f 3209 c1bc d2a6 a9d9 3f06 1e2c f970  o.2.......?..,.p
00000030: 1d90 ae31 bc5c 010d de8b 9a4b 6060 71b6  ...1.\.....K``q.
00000040: 71ce 43eb 505e 7759 42b9 e6c1 6bf5 06b9  q.C.P^wYB...k...
00000050: bd70 94fd 990f 2261 1257 76c2 7441 cbe0  .p...."a.Wv.tA..
00000060: 8538 8d9d 753e 4bd0 5c16 cb9c 57ea 8b62  .8..u>K.\...W..b
00000070: f804 76a2 d33b 7044 4ec7 93aa 56eb c0c1  ..v..;pDN...V...
```

# Programming using Public-Key Cryptography APIs

- Languages, such as Python, Java, and C/C++, have well-developed libraries that implement the low-level cryptographic primitives for public-key operations
- Python:
  - no built-in cryptographic library
  - use Python packages (e.g. PyCryptodome)
- We will cover:
  - Key Generation
  - Encryption and Decryption
  - Digital Signature

# Public-Key Cryptography APIs: Key Generation

- Python example (next slide) using Python Crypto APIs to generate a RSA key and save it to a file
- Lines in code:
  - Line (1): generate a 2048-bit RSA key
  - Line (2): export key() API serializes the key using the ASN.1 structure
  - Line (3): extract public-key component

# Public-Key Cryptography APIs: Key Generation (Contd.)

```
#!/usr/bin/python3

from Crypto.PublicKey import RSA

key = RSA.generate(2048)                                        ①
pem = key.export_key(format='PEM', passphrase='dees')           ②
f = open('private.pem','wb')
f.write(pem)
f.close()

pub = key.publickey()                                           ③
pub_pem = pub.export_key(format='PEM')
f = open('public.pem','wb')
f.write(pub_pem)
f.close()
```

# Public-Key Cryptography APIs: Encryption

- To encrypt a message using public keys, we need to decide what padding scheme
- For better security, it is recommended that OAEP is used
- Lines in code (example on next slide):
  - Line (1): import the public key from the public-key file
  - Line (2): create a cipher object using the public key

# Public-Key Cryptography APIs: Encryption (Contd.)

```
#!/usr/bin/python3

from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

message = b'A secret message!\n'

key = RSA.importKey(open('public.pem').read())  ①
cipher = PKCS1_OAEP.new(key)                    ②
ciphertext = cipher.encrypt(message)
f = open('ciphertext.bin','wb')
f.write(ciphertext)
f.close()
```

# Public-Key Cryptography APIs: Decryption

Uses the private key and the decrypt() API

```
#!/usr/bin/python3

from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

ciphertext = open('ciphertext.bin', 'rb').read()

prikey_pem = open('private.pem').read()
prikey = RSA.importKey(prikey_pem, passphrase='dees')
cipher = PKCS1_OAEP.new(prikey)
message = cipher.decrypt(ciphertext)
print(message)
```

# Public-Key Cryptography APIs: Digital Signature

- In Python code, one canuse PyCryptodome library’s Crypto.Signature package
- Four supported digital signature algorithms:
  - RSASSA-PKCS1-v1_5
  - RSASSA-PSS
  - DSA
  - RSASSA-PSS
- Show example with RSASSA-PSS

# Public-Key Cryptography APIs: Digital Signature using PSS

- Probabilistic Signature Scheme (PSS) is a cryptographic signature scheme designed by Mihir Bellare and Phillip Rogaway
- RSA-PSS is standardized as part of PKCS#1 v2.1
- Sign a message in combination with some random input.
- For same input:
  - two signatures are different
  - both can be used to verify

# Public-Key Cryptography APIs: Digital Signature using PSS (Contd.)

- Lines in code example:
  - line (1): create a signature object
  - line (2): generate the signature for the hash of a message

```
#!/usr/bin/python3

from Crypto.Signature import pss
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA

message = b'An important message'
key_pem = open('private.pem').read()
key = RSA.import_key(key_pem, passphrase='dees')
h = SHA256.new(message)
signer = pss.new(key)                          ①
signature = signer.sign(h)                     ②
open('signature.bin', 'wb').write(signature)
```

# Applications

We will cover:

- Authentication
- HTTPS and TLS/SSL
- Chip Technology Used in Credit Cards

# Applications: Authentication

- Typical way to conduct authentication is to use passwords
- Disadvantage:
  - A sends password to B: B can get hacked and A may use same password for multiple accounts
  - cannot be used for many parties to authenticate a single party
- Fundamental problem: password authentication depends on a shared secret

# Applications: Authentication (Contd.)

Solution:

- Making the encryption and decryption keys different
- generate the authentication data using one key, and verify the data using a different key

# Applications: Authentication (Contd.)

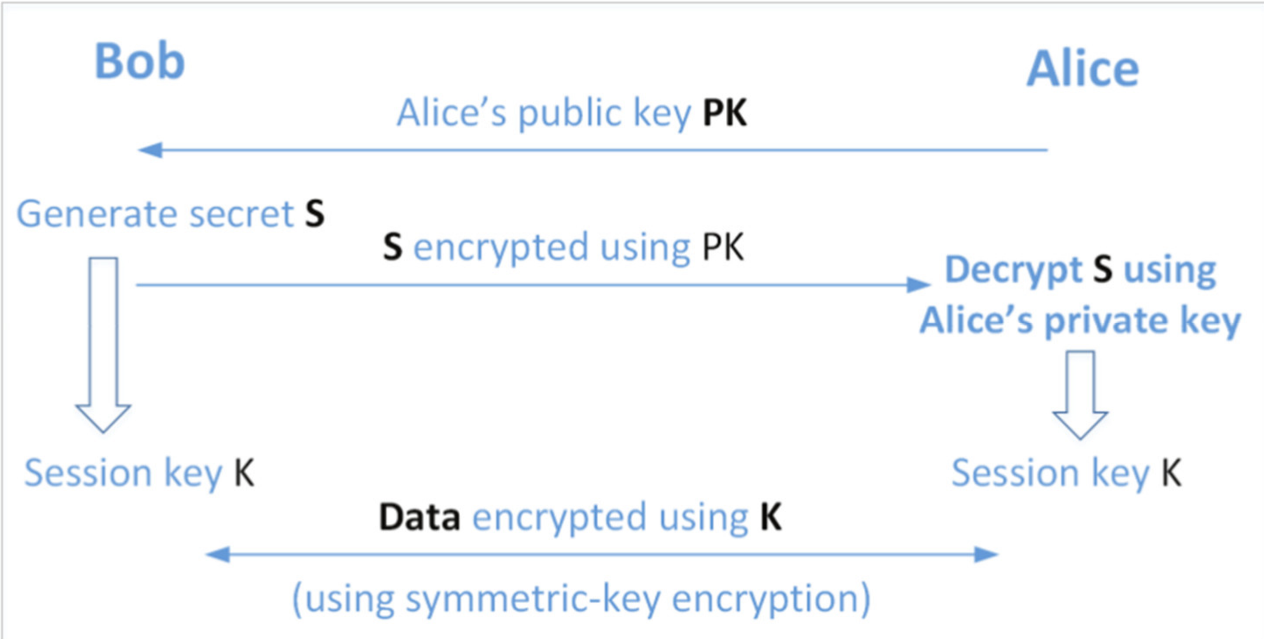
SSH Case Study

- SSH uses public-key based authentication to authenticate users
- Generate a pair of public and private keys: `ssh-keygen -t rsa`
  - private key: `/home/seed/.ssh/id_rsa`
  - public key: `/home/seed/.ssh/id_rsa.pub`
- For Server:
  - send the public key file to the remote server using a secure channel
  - add public key to the authorization file `~/.ssh/authorized_keys`
  - Server can use key to authenticate clients

# Applications: HTTPS and TLS/SSL

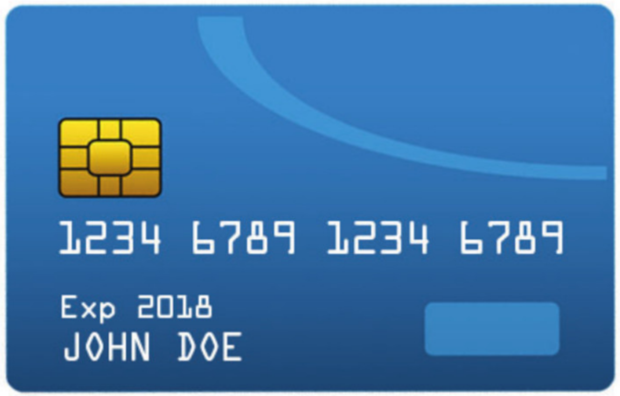
- HTTPS protocol is used to secure web services
- HTTPS is based on the TLS/SSL protocol (uses both public key encryption and signature
  - encryption using secret-key encryption algorithms
  - public key algorithms are mainly used for key exchange

# Applications: HTTPS and TLS/SSL (Contd.)



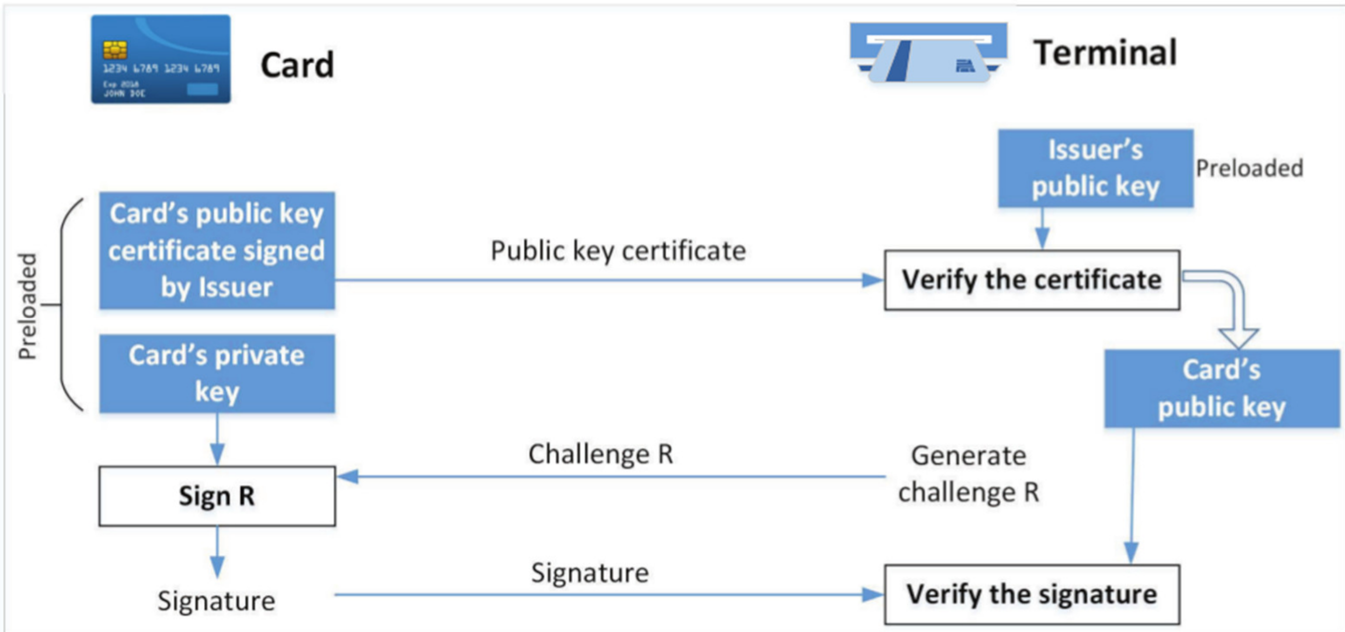
# Applications: Credit Card Chip

- Past: cards store card information in magnetic stripe (easy to clone)
- With Chip:
  - chips can conduct computations and store data (not disclosed to outside)
  - EMV standard (Europay, MasterCard, and Visa)
- We will cover how public key technologies are used for:
  - Card authentication
  - Transaction authentication



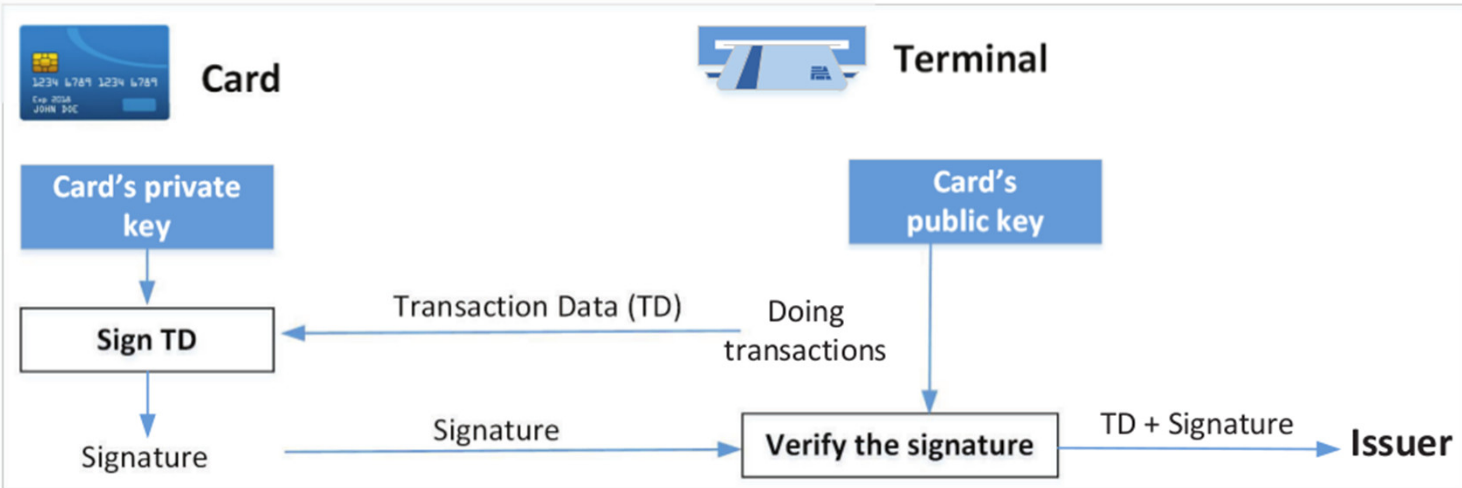
# Applications: Credit Card Chip Authentication

- Card contains a unique public and private key pair
  - Private key is protected and will never be disclosed to the outside
  - Public key is digitally signed by the issuer, so its authenticity can be verified by readers



# Applications: Credit Card Transaction Authentication

- Issuer needs to know whether the transaction is authentic
- Transaction needs to be signed by the card using its private key
- Verified Signature:
  - To issuers: card owner has approved the transaction
  - To honest vendor: enables the vendor to save the transactions and submit them later



# Summary

We covered:

- the basics of public key cryptography
- both theoretical and practical sides of public key cryptography
- RSA algorithm and the Diffie-Hellman Key Exchange
- tools and programming libraries to conduct public-key operations
- how public key is used in real-world applications