MTAT.07.017
Applied Cryptography

Public Key Cryptography
(Asymmetric Cryptography)

University of Tartu

Spring 2019
Diffie-Hellman Key Exchange

Ralph Merkle, Martin Hellman, Whitfield Diffie (1976)

- The first public-key algorithm
Diffie-Hellman (DH) Key Exchange

\[ (2^5)^4 = 2^{5 \cdot 4} = (2^4)^5 \]

- In practice multiplicative group of integers modulo \( p \) is used
- Discrete logarithm problem:
  - hard to find \( x \), given \( 2^x = 32 \mod p \)
- ElGamal and DSA based on DL problem
- Secure against passive eavesdropping
Adversary (Threat) Model

- What are the capabilities of the adversary?

Passive attacks (eavesdropping):

Active attacks (man-in-the-middle):

- Which attack is harder to execute?
- Without a threat model the word “secure” tells nothing
RSA

Adi Shamir, Ron Rivest, Leonard Adleman (1977)

- The most popular public-key cryptosystem
RSA algorithm

• Key generation:
  1. Choose two distinct prime numbers $p$ and $q$ (usually 1024-bits)
  2. Compute $n = pq$ (2048-bits)
  3. Compute $\varphi(n) = (p - 1)(q - 1)$
  4. Choose an integer $e$ such that $e$ and $\varphi(n)$ are coprime
  5. Find an integer $d$ such that $de \equiv 1 \mod \varphi(n)$

$n$ - modulus
$e$ - public exponent (encryption exponent)
$d$ - private exponent (decryption exponent)

Public key: $(n, e)$
Private key: $(d)$

• Encryption: $c \equiv m^e \mod n$
  
  Naive approach: $>>> m**e \% n$

• Decryption: $m \equiv c^d \mod n$
  
  Much faster: $>>> \text{pow}(m, e, n)$

• Integer factorization problem
RSA Encryption

The basis:

What is encrypted with one key can be decrypted only with the another and vice versa.

- What is encryption for?
  - Only the recipient could decrypt

- How do you encrypt?
  - Using public key of the recipient

- How does recipient decrypt?
  - Using his private key
RSA Signing

The concept of signing:

Encryption with private key, decryption with public key

- What is signing for?
  - Everyone could authenticate the origin

- How do you sign?
  - Encrypting with your private key

- How do others verify?
  - Decrypting with your public key

In practice message digest is encrypted (signed)
Public Key Cryptography

The benefits of asymmetric cryptography:

- Provides possibility for digital signatures
  - Data origin authentication
- Encryption key can be negotiated over non-confidential channel
  - Authenticated channel still required
Exponentiation

Which operation is harder to calculate: $x^{16384}$ or $x^{8191}$?

```python
>>> bin(16384)
'0b1000000000000000'
>>> bin(8191)
'0b11111111111111'
```

Number of multiplications: number of bits + number of “1” bits

Example:

$$x^8 = \left( x \cdot x \cdot y \cdot z \right) \quad x^7 = \left( x \cdot x \cdot y \cdot y \cdot x \right)$$
RSA Exponents

Key generation:

1. Choose two distinct prime numbers $p$ and $q$
2. Compute $n = pq$
3. Compute $\varphi(n) = (p - 1)(q - 1)$
4. Choose an integer $e$ such that $e$ and $\varphi(n)$ are coprime
5. Find an integer $d$ such that $de \equiv 1 \mod \varphi(n)$

- Can we choose $e$ that will provide faster encryption?
  - Yes! $e = 2^{16} + 1 = 65537$ (0b1000000000000001)
  - $e = 3$ (0b11) may also be used (not recommended)
  - Some implementations use random public exponent

- Can we instead choose $d$ that will provide faster decryption?
  - No! This will leak information about secret $d$
  - In fact, constant time exponentiation must be used for $d$
# Key Length Recommendations (NIST)

<table>
<thead>
<tr>
<th>Date</th>
<th>Minimum of Strength</th>
<th>Symmetric Algorithms</th>
<th>Asymmetric</th>
<th>Discrete Logarithm Key Group</th>
<th>Elliptique Curve</th>
<th>Hash (A)</th>
<th>Hash (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010 (Legacy)</td>
<td>80</td>
<td>2TDEA*</td>
<td>1024</td>
<td>160 1024</td>
<td>160</td>
<td>SHA-1**</td>
<td>SHA-1  SHA-224  SHA-256  SHA-384  SHA-512</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SHA-224  SHA-256  SHA-384  SHA-512</td>
<td></td>
</tr>
<tr>
<td>2011 - 2030</td>
<td>112</td>
<td>3TDEA</td>
<td>2048</td>
<td>224 2048</td>
<td>224</td>
<td>SHA-224  SHA-256  SHA-384  SHA-512</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SHA-224  SHA-256  SHA-384  SHA-512</td>
<td></td>
</tr>
<tr>
<td>&gt; 2030</td>
<td>128</td>
<td>AES-128</td>
<td>3072</td>
<td>256 3072</td>
<td>256</td>
<td>SHA-256  SHA-384  SHA-512</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SHA-224  SHA-256  SHA-384  SHA-512</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; 2030</td>
<td>192</td>
<td>AES-192</td>
<td>7680</td>
<td>384 7680</td>
<td>384</td>
<td>SHA-384  SHA-512</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SHA-224  SHA-256  SHA-384  SHA-512</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;&gt; 2030</td>
<td>256</td>
<td>AES-256</td>
<td>15360</td>
<td>512 15360</td>
<td>512</td>
<td>SHA-512</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SHA-224  SHA-256  SHA-384  SHA-512</td>
<td></td>
</tr>
</tbody>
</table>

http://www.keylength.com/
RSA private key file format

$ openssl genrsa -out priv.pem 1024
Generating RSA private key, 1024 bit long modulus
.......................++++++
.......................++++++
e is 65537 (0x10001)

$ cat priv.pem
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQDadIy9YfBPIk9Im1qOluderUItfj9FINQn/Gv+q+cHk06RgpphX6c+sIVvk/bEHmWGFvVcwMCXM5tzIeC/ns8sHu1b1IsUXSJwi0ArzuxPMBfay5TUu16oP/K0LBxYawa3xMSa+FmaAQsugBcWCYTM0iv+H7YkZdpDRZ++IbfWyX1wIDAQABAoGAQLdcZlJYVaktYa3Qh0VXSu2feGzrqr/+CeZ+u9CDPbxG/1Z4k7Y5npnAo0IVT
Z5Pp1sF4fjP9FXwM/SKUsbL6n/TR7U253KjxzfuBPMayjTMqqHTVDwbcJ0zhdG
emF1s3aZtRmZA8nvrooAQqhr5pfNcL/0i0mjf2+E4St3IxECQQD9rhVTm4NV1prrf2813zebpqbhzUPCbuK9/FmfZcx1Tg7EX6RPl9Fr0aTXQgCL9nGrZhKVraSSdwAZP
4q/oba0fAkEA3HP/hsUoi1m4VGXLOcDc+c5UYNcJkkmHJi7AAbzEO6FFrHyJhLC
9CsB40VayKbpxmAtN6Djhudltav/oFTgyQJaemsm1610GONTfLwm9vUmPsCvjjrntRbTTrta7/wqTdL2iNECQQCU9ufnB5Yxylu0RScYQ6ij4vXV5tD8buPgQsRhQ5xaqfzQvWQap0hR3FjOj7GcI1orvQcEgYOLFp7VyueqH56
-----END RSA PRIVATE KEY-----

PEM format (BASE64 encoded ASN.1 DER)

$ openssl rsa -in priv.pem -outform der -out priv.der
writing RSA key
RSA private key file format

$ dumpasn1 priv.der

0 605: SEQUENCE {
4  1:   INTEGER 0
7 129:   INTEGER
        :   00 DA 74 8C BD 61 F0 4F 22 4F 48 9A 5A B4 96 E0
        :   D7 ...
139  3:   INTEGER 65537
144 128:   INTEGER
        :   40 B7 5C 66 52 58 55 A9 2D 61 AD D0 87 45 57 4A
        :   97 ...
275  65:   INTEGER
        :   00 FD AE 15 53 9B 83 55 D6 9A EB 7F 6F 35 DF 37
        :   9F ...
342  65:   INTEGER
        :   00 DC 73 FF 86 C5 28 8B 59 B8 54 65 CB 5B 47 03
        :   C9 ...
409  64:   INTEGER
        :   7B 2B 26 D7 AD 4E 1B 43 53 7C BC 26 F6 F5 26 3E
        :   DD ...
475  65:   INTEGER
        :   00 87 56 C7 66 CB 9F 6A 7D 78 46 87 FF E2 57 A4
        :   D1 ...
542  65:   INTEGER
        :   00 94 F6 E7 E7 07 96 31 CA 5B 8E 45 27 18 43 A8
        :   7A ...
        :   }

14 / 25
RSA private key file format

--
-- Representation of RSA private key with information for
-- the CRT algorithm.
--
RSAPrivateKey ::= SEQUENCE {
   version Version,  
   modulus INTEGER, -- n
   publicExponent INTEGER, -- e
   privateExponent INTEGER, -- d
   prime1 INTEGER, -- p
   prime2 INTEGER, -- q
   exponent1 INTEGER, -- d mod (p-1)
   exponent2 INTEGER, -- d mod (q-1)
   coefficient INTEGER, -- (inverse of q) mod p
   otherPrimeInfos OtherPrimeInfos OPTIONAL
}

RSA public key file format

$ openssl rsa -inform der -in priv.der -pubout -outform der -out pub.der
writing RSA key
$ openssl rsa -inform der -in pub.der -pubin -out pub.pem
writing RSA key
$ dumpasn1 pub.der
  0 159: SEQUENCE {
    3 13: SEQUENCE {
      5 9: OBJECT IDENTIFIER rsaEncryption (1 2 840 113549 1 1 1)
      16 0: NULL
    :  }
  18 141: BIT STRING, encapsulates {
  22 137: SEQUENCE {
    25 129: INTEGER
      :  00 DA 74 8C BD 61 F0 4F 22 4F 48 9A 5A B4 96 E0
      :  C4 44 8B 5F 8F D1 48 35 09 FF 1A FF AA F9 C1 E4
      :  D3 A4 60 A6 98 57 E9 CF AC 22 F5 64 FD B1 07 99
      :  61 9F 57 00 9C C4 C1 79 B7 32 1E 0B F9 EC F2 C1
      :  EE D5 BD 48 B1 45 D2 27 08 8E 02 BC EE C4 F3 01
      :  7D AC B9 4D 4B B5 EA 83 FF 2B 49 41 C5 86 96 6B
      :  7C 4C 49 AF 85 99 A0 10 B2 E8 01 71 60 98 4C C3
      :  A2 BF E1 FB 62 46 5D A4 34 59 FB E2 1B 7D 6C 97
      :  D7
    157 3: INTEGER 65537
      :  }
      :  }
  :  }
RSA public key file format

SubjectPublicKeyInfo ::= SEQUENCE {
    algorithm   AlgorithmIdentifier,
    subjectPublicKey BIT STRING ::= RSAPublicKey
}

RSAPublicKey ::= SEQUENCE {
    modulus           INTEGER,  -- n
    publicExponent    INTEGER   -- e
}

AlgorithmIdentifier ::= SEQUENCE {
    algorithm OBJECT IDENTIFIER ::= rsaEncryption,
    parameters ANY DEFINED BY algorithm OPTIONAL ::= NULL
}

rsaEncryption OBJECT IDENTIFIER ::= { pkcs-1 1 }

Task: RSA utility – 7p

Implement RSA encryption and signing utility.

$ ./rsa.py

Usage:
encrypt <public key file> <plaintext file> <output ciphertext file>
decrypt <private key file> <ciphertext file> <output plaintext file>
sign <private key file> <file to sign> <signature output file>
verify <public key file> <signature file> <file to verify>

• Must support private and public keys in PEM and DER format (detect by content; use .decode(‘base64’))
• Halt if requested to encrypt plaintext larger than possible
• Must sign SHA256 hash (DigestInfo) of the file to sign
• Verification must output ”Verified OK” / ”Verification Failure”
• Encryption and signing according to PKCS#1 v1.5
• Use your own ASN.1 DER encoder and pyasn1
• Use pyasn1 version from Ubuntu packages (not from pip) (!)
RSA PKCS#1 v1.5

Encryption process:
1. Pad plaintext: 0x00 || 0x02 || PS || 0x00 || D
   - D – plaintext to encrypt
   - PS (padding string) – at least 8 random bytes (except 0x00)
   - Plaintext must be padded to size of modulus $n$
2. Convert padded byte string to integer
3. Calculate ciphertext: $c \equiv m^e \mod n$
   - in python: $c = \text{pow}(m, e, n)$
4. Convert ciphertext integer to byte string

Decryption process:
1. Convert ciphertext to integer
2. Calculate decryption: $m \equiv c^d \mod n$
3. Convert decrypted integer to byte string
4. Remove padding

RSA PKCS#1 v1.5

Signing process:
1. Construct plaintext (DER DigestInfo of the file to sign)
2. Pad plaintext: 0x00 || 0x01 || PS || 0x00 || D
   - D – plaintext to sign
   - PS (padding string) – zero or more 0xFF bytes
   - Plaintext must be padded to size of modulus $n$
3. Convert padded byte string to integer
4. Calculate signature: $s \equiv m^d \mod n$
5. Convert signature integer to byte string ($\text{length of } n$)

Verification process:
1. Convert signature byte string to integer
2. Calculate decryption: $m \equiv s^e \mod n$
3. Convert decrypted integer to byte string
4. Remove padding to obtain DigestInfo DER structure
5. Compare DigestInfo with DigestInfo of the signed file

Task: Test case

#!/bin/bash
echo "+] Generating RSA key pair..."
openssl genrsa -out priv.pem 1017
openssl rsa -in priv.pem -pubout -out pub.pem

echo "[+] Testing encryption..."
echo "hello" > plain.txt
./rsa.py encrypt pub.pem plain.txt enc.txt
openssl rsautl -decrypt -inkey priv.pem -in enc.txt -out dec.txt
diff -u plain.txt dec.txt

echo "[+] Testing decryption..."
openssl rsautl -encrypt -pubin -inkey pub.pem -in plain.txt -out enc.txt
./rsa.py decrypt priv.pem enc.txt dec.txt
diff -u plain.txt dec.txt

echo "[+] Testing signing..."
dd if=/dev/urandom of=filetosign bs=1M count=1
./rsa.py sign priv.pem filetosign signature
openssl dgst -sha256 -verify pub.pem -signature signature filetosign

echo "[+] Testing successful verification..."
openssl dgst -sha256 -sign priv.pem -out signature filetosign
./rsa.py verify pub.pem signature filetosign

echo "[+] Testing failed verification..."
openssl dgst -md5 -sign priv.pem -out signature filetosign
./rsa.py verify pub.pem signature filetosign
Verification Failure
Hybrid Encryption

How to encrypt plaintexts larger than modulus size?

- Increase modulus size
  - 2x modulus size increase – 8x slowdown
- Split in blocks and encrypt separately
  - Asymmetric encryption much slower than symmetric
- Use RSA to encrypt symmetric data encryption key:
Questions

• What does public key cryptography give us?
• How will you create RSA encrypted message to me?
• How will you verify my RSA signed message?
• What is hybrid encryption useful for?
• How are passive attacks different from active attacks?
• Why active attacks are harder to execute?
• What is threat model useful for?
• Why 2048-bit RSA does not have security level of 2048-bits?
• What will happen to cryptography the day quantum computers are invented?
Elliptic Curve Cryptography

\[ y^2 \pmod{p} = x^3 + ax^2 + bx + c \pmod{p} \]

- Curve defined by \(a, b, c, p\)
- Formulas for point addition, doubling
- DL problem: find \(x\), given \(a^x = b \pmod{p}\)
- EC DL problem: find \(x\), given \(P + P + \cdots + P = xP = T\)

- Security: 256-bit ECC \(\approx\) 3072-bit RSA
Estonian ID card (Infineon RSA keygen) flaw


Candidate prime $p$ constructed by:

$$p = k \times M + (65537^a \mod M)$$

- $M = 2 \times 3 \times 5 \times 7 \times 11 \times \ldots \times 701$ (971-bit constant)
- $k$ – 53-bit random number
- $a$ – 255-bit random number

Researchers found how to factor such $N$ using $2^{34.29}$ operations
- Each operation takes 212 ms on Intel Xeon E5-2650 v3 CPU

Moral: Do not invent your own crypto algorithms!