MH4311 Cryptography

Lecture 13
Public Key Encryption
Part 1: RSA

Wu Hongjun

Lecture Outline

- Classical ciphers
- Symmetric key encryption
- Hash function and Message Authentication Code
- **Public key encryption**
  - **RSA**
    - **Specification**
    - **Implementation**
    - **Security**
    - **Secure Implementation**
      - **p, q, n, d**
      - **Message padding (OAEP)**
      - **RSA Blinding**
  - ElGamal
- Digital signature
- Key establishment and management
- Introduction to other cryptographic topics

Recommended Reading

- CTP Section 5.1 to 5.7, Section 4.9
- HAC Section 8.1 and 8.2
- Wikipedia
  - Public key cryptosystem
    http://en.wikipedia.org/wiki/Public-key_cryptography
  - RSA
    http://en.wikipedia.org/wiki/RSA
  - Primality testing
    http://en.wikipedia.org/wiki/Primality_test
  - Integer factorization
    http://en.wikipedia.org/wiki/Integer_factorization
  - Optimal asymmetric encryption padding
    http://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding
  - PKCS
    http://en.wikipedia.org/wiki/PKCS

Why public key cryptosystem?

- Symmetric key encryption
  - The same secret key is used for encryption and decryption
- If the sender & receiver do not share a secret key, how to communicate secretly?
  - Common problem for large computer network
  - Public key cryptosystems are used to solve this problem
    - **Diffie-Hellman key exchange** (1976)
      - The first paper on public key cryptosystem (we will learn it later)
    - **Public key encryption**

Public Key Encryption

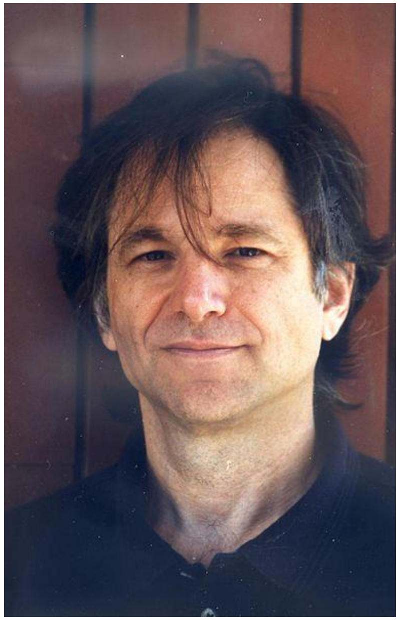
- Each receiver has two keys:
  - Encryption key (called public key)
    - Everyone knows the encryption key of a receiver
    - **Everyone can encrypt a message using the public key of a receiver**, then send the ciphertext to that receiver
  - Decryption key (called private key)
    - Only the receiver knows its decryption key
      - Difficult to derive the private key from public key
    - **Only the receiver can decrypt the ciphertext encrypted using its public key**

Public Key Encryption

- Many public key encryption algorithms
- RSA (1978)
  - The first public key encryption scheme
  - Based on the difficulty of integer factorization & ‘discrete logarithm’
- ElGamal (1985)
  - Based on the difficulty of discrete logarithm
    - discrete logarithm: $g^x \bmod p = y$

      (given y, difficult to find x when p is huge)

RSA

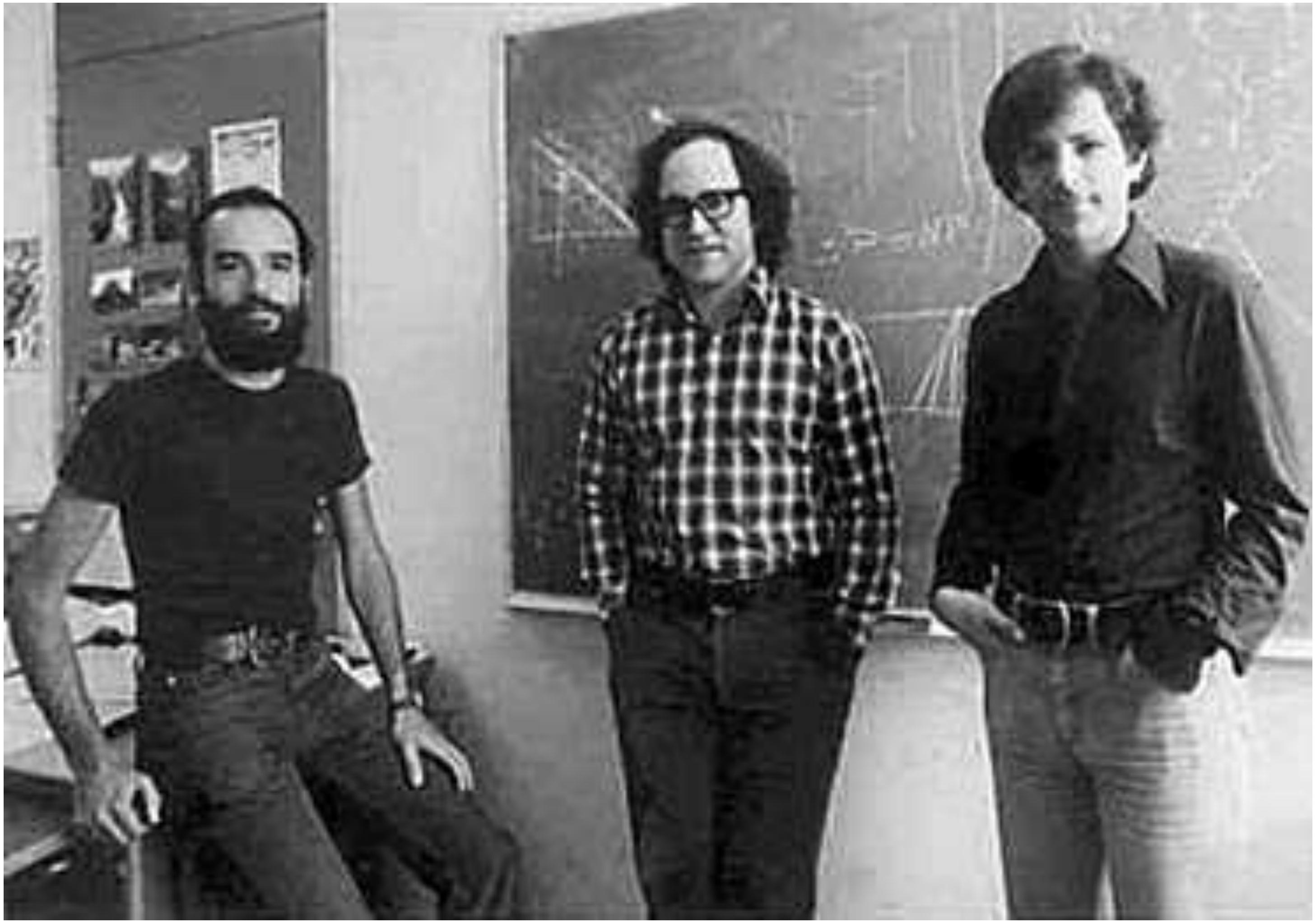


Ron **R**ivest

Adi **S**hamir

Leonard **A**dleman

RSA



RSA Specifications

Euler’s Totient Function

- Euler's totient function (also called phi function) gives the number of positive integers that are less than or equal to *n* and are coprime to *n*
- Denoted as $\varphi(n)$
- Example: $\varphi(7) = 6$ since 1, 2, 3, 4, 5 and 6 are coprime to 7
  $\varphi(6) = 2$ since 1 and 5 are coprime to 6
- $\varphi(n) = n \times \prod_{p_i|n}(1 - \frac{1}{p_i})$, where **each p_i is a distinct prime factor of *n***
  – Example:
  $$\varphi(7) = 7 \times \left(1 - \frac{1}{7}\right) = 6$$
  $$\varphi(6) = 6 \times \left(1 - \frac{1}{2}\right) \times \left(1 - \frac{1}{3}\right) = 2$$
  $$\varphi(12) = 12 \times \left(1 - \frac{1}{2}\right) \times \left(1 - \frac{1}{3}\right) = 4$$

RSA

- Key generation for each receiver:
  - Generate two different secret **large** prime numbers p and q
  - Compute $n = p \times q$
  - Compute $\varphi(n) = (p\text{-}1) \times (q\text{-}1)$
  - Choose an integer e which is coprime to $\varphi(n)$
  - Find d satisfying $e \times d \equiv 1 \pmod{\varphi(n)}$

public key: e, n

private key: d

RSA

- Encryption

$$\boldsymbol{c = m^e \bmod n}$$ (plaintext m: $0 < m <$ n)

- Decryption

$$\boldsymbol{m = c^d \bmod n}$$

RSA Decryption Recovers Message

Simple but incomplete proof:

- Euler's theorem:
  Let a be a positive integer coprime to n, then
  $$a^{\varphi(n)} \equiv 1 \pmod{n}$$
- RSA decryption:
  $$\begin{aligned} c^d \bmod n &= (m^e \bmod n)^d \bmod n \\ &= m^{ed} \bmod n \\ &= m^{\beta\varphi(n)+1} \bmod n \end{aligned}$$
  If m and n are coprime, then $m^{\beta\varphi(n)} \bmod n = 1$,
  so $c^d \bmod n = m$.

RSA Decryption Recovers Message

The complete proof that RSA decryption is the inverse of encryption requires the following theorems:

- Fermat's little theorem:

Let *a* be a positive integer coprime to a prime number *p*, then

$$a^{p-1} \equiv 1 \pmod{p}$$

RSA Decryption Recovers Message

The complete proof requires the following theorems:

- Chinese Remainder Theorem (special case):

If n_1 and n_2 are coprime, a and x are positive integers less than $n_1 n_2$,

$$x \equiv a \pmod{n_1}$$

$$x \equiv a \pmod{n_2}$$

then there is a unique solution

$$x = a$$

Brief explanation:

$n_1 \mid (x-a)$

$n_2 \mid (x-a)$

Since n_1 and n_2 are coprime, we get

$n_1 n_2 \mid (x-a)$

i.e., $x\text{-}a \bmod n_1 n_2 = 0$

RSA Decryption Recovers Message

- complete proof:

Let $x = c^d \bmod n$,

$$
\begin{aligned}
x \bmod p &= ((m^e)^d \bmod n) \bmod p \\
&= (m^e)^d \bmod p \\
&= m^{\beta(p-1)(q-1)+1} \bmod p
\end{aligned}
$$

If m and p are coprime, according to Fermat's little theorem: $m^{p-1} \bmod p = 1$

$$\therefore \quad x \bmod p = m^{\beta(p-1)(q-1)+1} \bmod p = m \bmod p \qquad (1)$$

If m is a multiple of p, then

$$x \bmod p = m^{\beta(p-1)(q-1)+1} \bmod p = 0 = m \bmod p \qquad (2)$$

From (1) and (2), $\quad x \equiv m \pmod{p} \qquad (3)$

Similarly: $\quad x \equiv m \pmod{q} \qquad (4)$

From (3), (4) and Chinese Remainder Theorem:

$$x = m \bmod pq = m$$

RSA

- Example (Toy RSA):

  Key generation:

  - $p = 61$, $q = 53$
  - $n = 61 \times 53 = 3233$
  - $\varphi(n) = (61\text{-}1)(53\text{-}1) = 3120$
  - choose public key $e = 17$, e is coprime to $\varphi(n)$
  - find private key $d = 2753$ satisfying $e \times d \equiv 1 \bmod \varphi(n)$

  Encryption:

  If $m = 37$, then $c = 37^{17} \bmod 3233 = 1350$

  Decryption:

  $m = 1350^{2753} \bmod 3233 = 37$

RSA Implementation

RSA Implementation

- Key generation:
  - Generate two distinct large prime numbers ***p* and *q***
  - Compute $n = p \times q$
  - Compute $\varphi(n) = (p\text{-}1) \times (q\text{-}1)$
    - φ is Euler's totient function
  - Choose an integer e that is coprime to $\varphi(n)$
  - Find ***d*** satisfying $e \times d \equiv 1 \bmod \varphi(n)$

- Encryption

$$c = \mathbf{m^e \bmod n}$$

- Decryption

$$m = \mathbf{c^d \bmod n}$$

RSA Implementation

To implement RSA:

1. How to generate two large prime numbers p & q?
2. How to compute (m^e mod n) and (c^d mod n) efficiently?

RSA Implementation

To Implement RSA:

1. **How to generate two large prime numbers *p* & *q*?**
2. How to compute (m^e mod n) and (c^d mod n) efficiently?

Generate large prime number

- To generate a large prime number, we normally use the following approach:
  - Generate a random large integer
  - Then test whether it is prime or not
  - Repeat the above two steps until we find a prime
- Questions:
  - How to generate a large random integer? (will learn later)
  - How to test whether a large random integer is prime?
  - How many integers should be tested (on average) in order to find a prime?

Generate large prime number

- Questions on generating large prime numbers:
  - How to generate a large random integer? (will learn later)
  - **How to test whether a large random integer is prime?**
  - How many integers should be tested (on average) in order to find a prime?

Generate large prime number: Primality test

- Primality tests
  - Naïve method
  - Probabilistic tests
    - Low complexity
    - Commonly used
  - Fast deterministic tests

Generate large prime number: Primality test

- Naïve primality test
  - The simplest primality test
  - To test whether an integer n is prime or not, try all the integers less than or equal to $n^{0.5}$ to check whether n is divisible by any of those integers
  - Complexity: $O(n^{0.5})$
    - Too high for large integers

Generate large prime number: Primality test

- Probabilistic primality tests
  - Many probabilistic primality tests
  - Fermat primality test
    - Simple, but not useful for detecting some special composite numbers
    - Useful for quick screening, then test the remaining numbers using other primality testing methods
  - Miller-Rabin primality test
    - The commonly used primality testing method
      - Mathematica, OpenSSL, …

Generate large prime number: Primality test

- Fermat primality test
  - Based on Fermat's little theorem:

    $a^{p-1} \bmod p = 1$ if p is prime, and a is coprime to p.
  - To test whether an integer n is prime or not, choose many integers a less than n and larger than 1,
    - If $a^{n-1} \bmod n \neq 1$, then n is composite
    - If $a^{n-1} \bmod n = 1$, then n may or may not be prime
  - As more values of a are tested, the accuracy of primality test improves
    - But for some special composite number n (called Carmichael numbers, or Fermat pseudoprimes), primality test does not work, since for all the a coprime to n,

      $a^{n-1} \bmod n = 1$
    - Some Carmichael numbers: 561, 1105, 1729, 2465, 2821, 6601, ….

Generate large prime number: Primality test

- Miller-Rabin primality test:

Given an integer n, write $n-1=2^r s$, where s is odd

Choose a random integer a with $2 \le a \le n\text{-}1$

1. If $a^s \not\equiv 1 \pmod{n}$ and $a^{2^j s} \not\equiv -1 \pmod{n}$ for all $0 \le j \le r-1$, then n is a composite;
2. Otherwise, n may or may not be prime

If n is identified as composite, stop the test.
If n is not identified as composite, choose a different integer a and repeat the test. (How many values of a should be tested?)

Generate large prime number: Primality test

- A prime can always pass the Miller-Rabin primality test
  - Suppose that n is a prime. Let $n-1=2^r s$, where s is odd
  - For every positive integer a (a is less than n), we have

$$a^{n\text{-}1} \equiv 1 \pmod{n}$$

  - Now we keep taking square root of $a^{n-1} \pmod{n}$,

1) If we get -1, it means that

$$a^{2^j s} \equiv -1 \pmod{n} \text{ for some } 0 \le j \le r-1.$$

2) If we never get -1 after taking out every power of 2, we are left with

$$a^s \equiv 1 \pmod{n}$$

Generate large prime number: Primality test

- Every composite number can eventually be detected in the Miller-Rabin primality test
  - Proved by Rabin in 1977.

Generate large prime number: Primality test

- Miller-Rabin primality test
  - A prime can always pass the above test (will never be identified as composite)
  - **After testing *N* random distinct integers *a*, a composite number is not identified as composite with probability less than 2^{-2N}**
    - To test whether a 1024-bit number is prime or not, 64 trials are sufficient for high accuracy

Generate large prime number: Primality test

- Miller-Rabin primality test example

  Determine whether $n = 221$ is prime.

  Step 1. $n - 1 = 220 = 2^2 \times 55$, so $r = 2$, $s = 55$.

  Step 2. select a number $a = 174$.

  - $a^s \bmod n = 174^{55} \bmod 221 = 47 \neq 1$
  - $a^s \bmod n = 174^{55} \bmod 221 = 47 \neq n - 1$
  - $a^{2s} \bmod n = 174^{110} \bmod 221 = 220 = n - 1$ **=> *n* may be prime**

  Step 3. select a number $a = 137$

  - $a^s \bmod n = 137^{55} \bmod 221 = 188 \neq 1$
  - $a^s \bmod n = 137^{55} \bmod 221 = 188 \neq n - 1$
  - $a^{2s} \bmod n = 137^{110} \bmod 220 = 205 \neq n - 1$

  => *n* is a composite

Generate large prime number: Primality test

- Fast deterministic primality tests
  - In 2002, Agrawal, Kayal and Saxena found a new deterministic primality test (AKS), with complexity $O((\log n)^{12})$
  - In 2005, the complexity is reduced to $O((\log n)^{6})$
    - The complexity of deterministic tests is too high for practical applications (for example, it is too expensive to find a 1024-bit prime using the deterministic primality test.)

Generate large prime number

- Questions on generating large prime numbers:
  - How to generate a large random integer? (will learn later)
  - How to test whether a large random integer is prime?
  - **How many integers should be tested (on average) in order to find a prime?**

Generate large prime number: Prime Distribution

$\pi(x)$: the number of primes less than or equal to a real number x

- Prime Distribution Theorem

$$\lim_{x \to \infty} \frac{\pi(x)}{x} = \frac{1}{\ln(x)}$$

Generate large prime number: Prime Distribution

$$\lim_{x\to\infty} \frac{\pi(x)}{x} = \frac{1}{\ln(x)}$$

- A random 1024-bit integer is prime with probability about

$$\frac{1}{\ln 2^{1024}} \approx \frac{1}{710}$$

- A random 2048-bit integer is prime with probability about

$$\frac{1}{\ln 2^{2048}} \approx \frac{1}{1420}$$

$\Rightarrow$ The probability that a large random integer is prime is sufficiently large for practical applications

Generate large prime number

- For the current cryptography applications, we generate primes between 1024 bits and 8192 bits
  - Miller-Rabin primality test can be applied to generate such a prime efficiently
- As of 2018, the largest prime number being identified is 77,232,917 bits (it is $2^{77,232,917}-1$)
  - Project "Great Internet Mersenne Prime Search"
  - The computational cost is extremely high for finding a HUGE prime
    - the cost of testing whether a huge number is prime is high
    - the density of the primes is low for huge numbers

RSA Implementation

To Implement RSA:

1. How to generate two large prime numbers p & q?
2. **How to compute (m^e mod n) and (c^d mod n) efficiently?**

Compute modular exponentiation

- The exponentiation by squaring is a general method for fast computation of modular exponentiation
  - Some variants are called square-and-multiply algorithm or binary exponentiation
  - Computing (a^x mod n) takes $O(\log x)$ modular multiplications

Compute modular exponentiation

Basic idea of exponentiation by squaring :

1. Represent a t-bit exponent x in binary format as

$$x = x_{t-1}x_{t-2}\cdots x_2 x_1 x_0, \text{ i.e., } x = \sum_{i=0}^{t-1} x_i 2^i$$

2. Compute $y_i = a^{2^i} \bmod n$ as

t-*1* square-mod operations

$$\underline{y_i = (y_{i-1})^2 \bmod n}, \text{ where } y_0 = a^{2^0} \bmod n = a$$

3. Then $a^x \bmod n$ is computed efficiently as

$$a^x \bmod n = a^{\sum_{i=0}^{t-1} x_i 2^i} \bmod n = (\prod_{i=0}^{t-1} a^{x_i 2^i}) \bmod n$$

$$= (\prod_{i=0}^{t-1} (a^{2^i})^{x_i}) \bmod n = \underline{(\prod_{i=0}^{t-1} y_i^{x_i}) \bmod n}$$

At most *t*-1 multiply-mod operations

Compute modular exponentiation

- Implement the method on the previous slide using the right-to-left square-and-multiply algorithm

$$y = a,\ z = 1$$

$$\text{for } i = 0 \text{ to } t-1 \text{ do}$$

$$\{$$

$$\quad \text{if } x_i = 1, \text{ then } z = z \cdot y \bmod n$$

$$\quad y = y^2 \bmod n$$

$$\}$$

Compute modular exponentiation

- Fast modular exponentiation can also be implemented using the left-to-right square-and-multiply algorithm:

$$
\begin{aligned}
& z = 1 \\
& \text{for } i = t-1 \text{ downto } 0 \text{ do} \\
& \{ \\
& \qquad z = z^2 \bmod n \\
& \qquad \text{if } x_i = 1, \text{ then } z = z \cdot a \bmod n \\
& \}
\end{aligned}
$$

Compute modular exponentiation

- Example of right-to-left square-and-multiply algorithm:

$$23^{20} \bmod 29 = ?$$

20 in binary format is 10100

Initializing: $z = 1, y = 23$

1010**0** : $y = y^2 \bmod 29 = 7$

101**0**0 : $y = y^2 \bmod 29 = 7^2 \bmod 29 = 20$

10**1**00 : $z = z \times y = 1 \times 20 = 20$

$y = y^2 \bmod 29 = 20^2 \bmod 29 = 23$

1**0**100 : $y = y^2 \bmod 29 = 23^2 \bmod 29 = 7$

10100 : $z = z \times y = 20 \times 7 \bmod 29 = 24$

$\cancel{y = y^2 \bmod 29 = 7^2 \bmod 29 = 20}$

$\Longrightarrow 23^{20} \bmod 29 = 24$

Compute modular exponentiation

- Example of left-to-right square-and-multiply algorithm:

  $23^{20} \bmod 29 = ?$

20 in binary format is 10100

Initializing: $z = 1$

10100 : $z = z^2 \bmod 29 = 1$

$z = z \times 23 = 23$

1**0**100 : $z = z^2 \bmod 29 = 23^2 \bmod 29 = 7$

10**1**00 : $z = z^2 \bmod 29 = 7^2 \bmod 29 = 20$

$z = z \times 23 = 20 \times 23 \bmod 29 = 25$

101**0**0 : $z = z^2 \bmod 29 = 25^2 \bmod 29 = 16$

1010**0** : $z = z^2 \bmod 29 = 16^2 \bmod 29 = 24$

==> $23^{20} \bmod 29 = 24$

RSA Security

RSA Security

Attacks on RSA:

- **To factorize *n***
    - Once *n* is factorized, *d* can be computed
    - Difficult for large *n*
- **Other attacks**

RSA Security: Integer Factorization

- Integer factorization
  - Here we consider only **RSA moduli**
    - Product of two primes (also called semiprimes, biprimes)
- Many integer factorization techniques
  - Trial division
  - …..
  - Dixon’s random squares algorithm
    - Quadratic sieve
    - General number field sieve

RSA Security: Integer Factorization

- Trial division
  - To factorize integer n, try all the integers less than or equal to $n^{0.5}$ to check whether n is divisible
  - Complexity: $O(n^{0.5})$

RSA Security: Integer Factorization

- Fermat's factorization method:
  - Many factorization techniques are based on this method.

If we can find $x \not\equiv \pm y \pmod{n}$ and $x^2 \equiv y^2 \pmod{n}$, we can factorize n:

$x^2 \equiv y^2 \pmod{n}$ => $n \mid (x-y)(x+y)$ => n is a factor of $(x-y)(x+y)$;

$x \not\equiv y \pmod{n}$ => $n \nmid x-y$ => n is not a factor of $x-y$

$x \not\equiv -y \pmod{n}$ => $n \nmid x+y$ => n is not a factor of $x+y$

Therefore one factor of n is factor of $x-y$,

another factor of n is a factor of $x+y$

So $\gcd(x-y, n)$ is a non-trivial factor of n,

$\gcd(x+y, n)$ is another non-trivial factor of n

Example: $10^2 \equiv 32^2 \pmod{77}$ => $\gcd(10+32, 77) = 7$, $\gcd(10-32, 77) = 11$

RSA Security: Integer Factorization

- Dixon's random squares algorithm

  Smooth number:
  - An integer which factors completely into small prime numbers
  - A positive integer is called ***B*-smooth** if none of its prime factors is greater than *B*.
  - Example:

    $1620 = 2^2 \times 3^4 \times 5$

    1620 is 5-smooth since none of its prime factors is greater than 5.

RSA Security: Integer Factorization

- Dixon's random squares algorithm (cont.)

$Q(x) \approx \alpha\sqrt{n}$

1) Let $m = \lfloor \sqrt{n} \rfloor$, define a function $Q(x) = (m+x)^2 - n$

2) Choose $B \approx 2^{(\log_2 n)^{1/2} (\log_2 \log_2 n)^{1/2}}$ (derivation given in textbook)

   Denote those primes not larger than B as $\{p_1, p_2, \cdots, p_t\} = \{-1, 2, 3, 5, \cdots, p_t\}$

3) Randomly select small (positive or negative) integers x.

   Keep those integers satisfying that $Q(x)$ is B-smooth,

   and denote them as $x_1, x_2, \cdots, x_\mu$

$$Q(x_i) = p_1^{e_{i,1}} \times p_2^{e_{i,2}} \times p_3^{e_{i,3}} \times \cdots \times p_t^{e_{i,t}}$$

RSA Security: Integer Factorization

4) With t such x_i, we can find a subset $A \subset \{1,2,3,4,5,\cdots,t\}$,

so that $\sum_{i \in A} e_{i,j}$ is even for all the values of j $(1 \le j \le t)$

(solving binary linear equations)

5) Then $$\prod_{i \in A} Q(x_i) = p_1^{\sum_{i \in A} e_{i,1}} \times p_2^{\sum_{i \in A} e_{i,2}} \times p_3^{\sum_{i \in A} e_{i,3}} \times \cdots \times p_t^{\sum_{i \in A} e_{i,t}}$$

$$\prod_{i \in A} Q(x_i) = y^2, \text{ where } y = p_1^{(\sum_{i \in A} e_{i,1})/2} \times p_2^{(\sum_{i \in A} e_{i,2})/2} \times p_3^{(\sum_{i \in A} e_{i,3})/2} \times \cdots \times p_t^{(\sum_{i \in A} e_{i,t})/2}$$

$$\left(\prod_{i \in A} (m + x_i)\right)^2 \equiv y^2 \pmod{n}$$

Let $z = \prod_{i \in A} (m + x_i)$

$$z^2 \equiv y^2 \pmod{n}$$

If $z \not\equiv \pm y \bmod n$, then $\gcd(z - y, n)$ gives a factor of n

Example:

Factorize

$n = 4841$

1. $m = \lfloor \sqrt{n} \rfloor = 69, Q(x) = (m+x)^2 - n$

2. set $B = 11$, the factor base is $\{-1,2,3,5,7,11\}$

3. $x = -8 \quad \rightarrow \quad Q(x) = -1120 = (-1) \times 2^5 \times 5 \times 7$

$x = -4 \quad \rightarrow \quad Q(x) = -616 = (-1) \times 2^3 \times 7 \times 11$

$x = -2 \quad \rightarrow \quad Q(x) = -352 = (-1) \times 2^5 \times 11$

$x = 0 \quad \rightarrow \quad Q(x) = -80 = (-1) \times 2^4 \times 5$

$x = 2 \quad \rightarrow \quad Q(x) = 200 = 2^3 \times 5^2$

$x = 3 \quad \rightarrow \quad Q(x) = 343 = 7^3$

4. We now solve the linear binary equations:

(-1)'s exponents modulo 2

....

5's exponents modulo 2

....

11's exponents modulo 2

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} a1 \\ a2 \\ a3 \\ a4 \\ a5 \\ a6 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \Rightarrow \begin{aligned} a_1 &= a_5 \\ a_2 &= a_5 + a_6 \\ a_3 &= a_5 + a_6 \\ a_4 &= a_5 \end{aligned}$$

$\{0,1,1,0,0,1\}$ is a solution => The set A can be: $\{x = -4, x = -2, x = 3\}$

Example:

Factorize

$n = 4841$

5. $y^2 = Q(-4)\times Q(-2)\times Q(3) = (-1)^2 \times 2^8 \times 7^4 \times 11^2$

$\Rightarrow\ y = (-1)\times 2^4 \times 7^2 \times 11 \equiv -3783 \pmod{4841}$

$z = (m-4)\times(m-2)\times(m+3) \equiv 3736 \pmod{4841}$

$\gcd(z-y,n) = \gcd(3736+3783, 4841) = 103$

$\gcd(z+y,n) = \gcd(3736-3783, 4841) = 47$

$n = 4841 = 47\times 103$

RSA Security: Integer Factorization

- Quadratic sieve
  - Very similar to Dixon's random squares algorithm
  - But with efficient sieving method to generate smooth numbers
- General number field sieve
  - Improve the quadratic sieve
  - Convert the integer factorization problem to factorization over algebraic number field
    - so as to generate "more" smooth numbers

RSA Security: Integer Factorization

- Complexities of factorization algorithms

Trial division: $O(\sqrt{n}) \rightarrow O(e^{0.5\underline{\ln n}})$

Dixon's Random Squares Algorithm: $O(e^{(1+O(1))\underline{(\ln n)^{1/2}}(\ln\ln n)^{1/2}})$

Quadratic sieve: $O(e^{(1+O(1))\underline{(\ln n)^{1/2}}(\ln\ln n)^{1/2}})$

General number field sieve: $O(e^{(1.92+O(1))\underline{(\ln n)^{1/3}}(\ln\ln n)^{2/3}})$

for 2048-bit n, $0.5\ln n$ is about 709.8;

$(\ln n)^{1/2}(\ln\ln n)^{1/2}$ is about 101.5

$(\ln n)^{1/3}(\ln\ln n)^{2/3}$ is about 42.1

RSA Security: Integer Factorization

- The size of RSA moduli

  NIST recommendation, 2007:

size of *n*	**security level**
1024-bit	80 bits
2048-bit	112 bits
3072-bit	128 bits
7680-bit	192 bits
15360-bit	256 bits

RSA Security: Integer Factorization

- Some RSA modulus factorization records
  - 129-digit modulus 1994 → Quadratic sieve
  - 155-digit (512-bit) modulus 1999
  - 174-digit (576-bit) modulus 2003
  - 200-digit (663-bit) modulus 2005
  - 232-digit (768-bit) modulus 2009

General number field sieve

Complexity: about 2000 CPU cores (2.2GHz) for 1 year

1024-bit modulus: when? method?

RSA Security: Integer Factorization

- On quantum computer, factorization is an easy problem (Shor, 1995)
  - But it is unknown when a practical quantum computer can be built

RSA Security: Trivial Attacks

- Trivial attacks
  - If p or q is known to the attacker → broken
  - If $\varphi(n)$ is known to the attacker → broken

RSA Security: Shared Modulus

- Attack on shared modulus
  - Shared modulus
    - each user is given a public key (e_i, n) and private key d_i
    - They share the same modulus n
  - Attack
    - Each user can factorize n easily from e_i and d_i
    - Then each user can find the private keys of other users
    - How to factorize?

RSA Security: Shared Modulus

- Attack on shared modulus (cont.)
  - Factorize n from e and d

1) Since $e \cdot d \equiv 1 \pmod{\varphi(n)}$,

$$e \cdot d - 1 = \beta(p-1)(q-1),$$

we know that $e \cdot d - 1$ is even

2) Select an integer a ($a < n$), compute

$$y = a^{(e \cdot d - 1)/2} \bmod n$$

3) We know that $a^{e \cdot d - 1} \bmod n = a^{\beta\varphi(n)} \bmod n = 1$ (Euler's theorem)

4) From 2) and 3), we know that

$$y^2 \equiv 1 \pmod{n}$$

Thus $\gcd(y-1, n)$ gives a factor of n if $y \not\equiv \pm 1 \pmod{n}$

RSA Security: Small encryption exponent

- The exponent e is too small
  - Attack 1:

    Example: if $e = 3$, then for small m (say, $m < n^{1/3}$),

    $$c = m^3 \bmod n = m^3$$

    => m can be recovered from c easily

RSA Security: Small encryption exponent

- The exponent e is too small
  - Attack 2:

    Example: if e = 3, and m is large. The same message m was sent to 3 different receivers

$$c_1 = m^3 \bmod n_1 \quad (1)$$

$$c_2 = m^3 \bmod n_2 \quad (2)$$

$$c_3 = m^3 \bmod n_3 \quad (3)$$

From Chinese Remainder Theorem and (1), (2), (3), $m^3 \bmod n_1n_2n_3$ can be obtained, i.e., m^3 becomes known. m can thus be recovered easily from m^3

RSA Security: Small encryption exponent

- Chinese Remainder Theorem
  - If the integers n_i are coprime to each other, there exists an integer x satisfying

$$x \equiv a_1 (\mathrm{mod}\, n_1)$$
$$x \equiv a_2 (\mathrm{mod}\, n_2)$$
$$\cdots\cdots$$
$$x = a_k (\mathrm{mod}\, n_k)$$

  - Let $N = n_1 n_2 \cdots n_k,\ \ N_i = N / n_i$.

    Apply the extended Euclidean algorithm to find M_i which is the multiplicative inverse of N_i modulo n_i .

    Then the solution is: $x \equiv \sum_{i=1}^{k} a_i M_i N_i \ \ (\mathrm{mod}\ \ N)$

RSA Security: Small encryption exponent

- Recommended value: $e = 65537 = 2^{16}+1$
  - Encryption takes 17 modular multiplications
  - Fast encryption, but slow decryption
    - Encryption is about 80 times faster than decryption for 1024-bit n
    - But RSA encryption with this e and 1024-bit n is more than 50 times slower than AES encryption on computer

RSA Security: Small decryption component

- How about choose small private key d to increase decryption speed?
  - In the key generation process, choose d first, then compute e
  - But **the value of *d* must be large for security reason**
    - Brute force attack: the size of d should be more than 128 bits
    - Advanced attack:
      - If $\mathrm{d} < \mathrm{n}^{0.25}$, d can be recovered from e and n easily (1987)
      - If $d < n^{0.292}$, d can be recovered from e and n easily (1998)
      - It is conjectured that if $d < n^{0.5}$, d can be recovered from e and n easily

RSA Security: Low-entropy plaintext

- Attack on plaintext with low entropy
  - Two properties are used in the attack
    - ‘Public’ encryption: everyone can perform the encryption of any message
    - deterministic encryption: the same plaintext is always encrypted to the same ciphertext
  - For plaintext with small entropy, the attacker can encrypt those possible messages, then compare the ciphertexts with the received ciphertext to recover the plaintext
  - A more advanced attack is given in the next slide

RSA Security: Low-entropy plaintext

- Example: the message size is small

  Attack: A 64-bit secret m is encrypted as $c = m^e \bmod n$

  (n, e, d are huge)

  With probability about 20%, a random 64-bit m can be written as $m = m_1 m_2$, where $m_1, m_2 < 2^{34}$.

  Now an attacker builds two tables:

$$T_1[i] = \frac{c}{i^e} \bmod n \quad \text{for } 1 \le i \le 2^{34}$$

$$T_2[j] = j^e \bmod n \quad \text{for } 1 \le j \le 2^{34}$$

If $T_1[i] = T_2[j]$ for some i and j, then the message $m = i \times j$

Attack complexity: about 2×2^{34}

RSA Secure Implementation

RSA Secure Implementation

- Large modulus
  - 3072 bits for 128-bit security
  - 15360 bits for 256-bit security
- Generate p and q independently and randomly
- Private key larger than $n^{0.5}$
- Message padding
  - **To introduce secret and random info to encryption**
  - To pad message m so that the length of padded message is close to that of n
- RSA blinding
  - **To introduce secret and random info to decryption**

RSA Message Padding

- "Textbook" RSA Encryption

  $\boldsymbol{c = m^e \bmod n}$ (plaintext m: $0 < m < n$)

- Risk
  - Encryption algorithm is **deterministic** and public: The same plaintext is always encrypted to the same ciphertext
    - If there is insufficient entropy in a plaintext, it may be possible to encrypt many possible plaintexts so as to identify the plaintext from the ciphertext
    - If the public key size is small, there may be no modulation operation

RSA Message Padding

- **Never use the "textbook" RSA in practice**

- Padding is necessary
  - Pad the message to large size
  - Introduce secret and random information to encryption

- The RSA message padding used in SSL (before 1998) is insecure
  - It leaks the private key

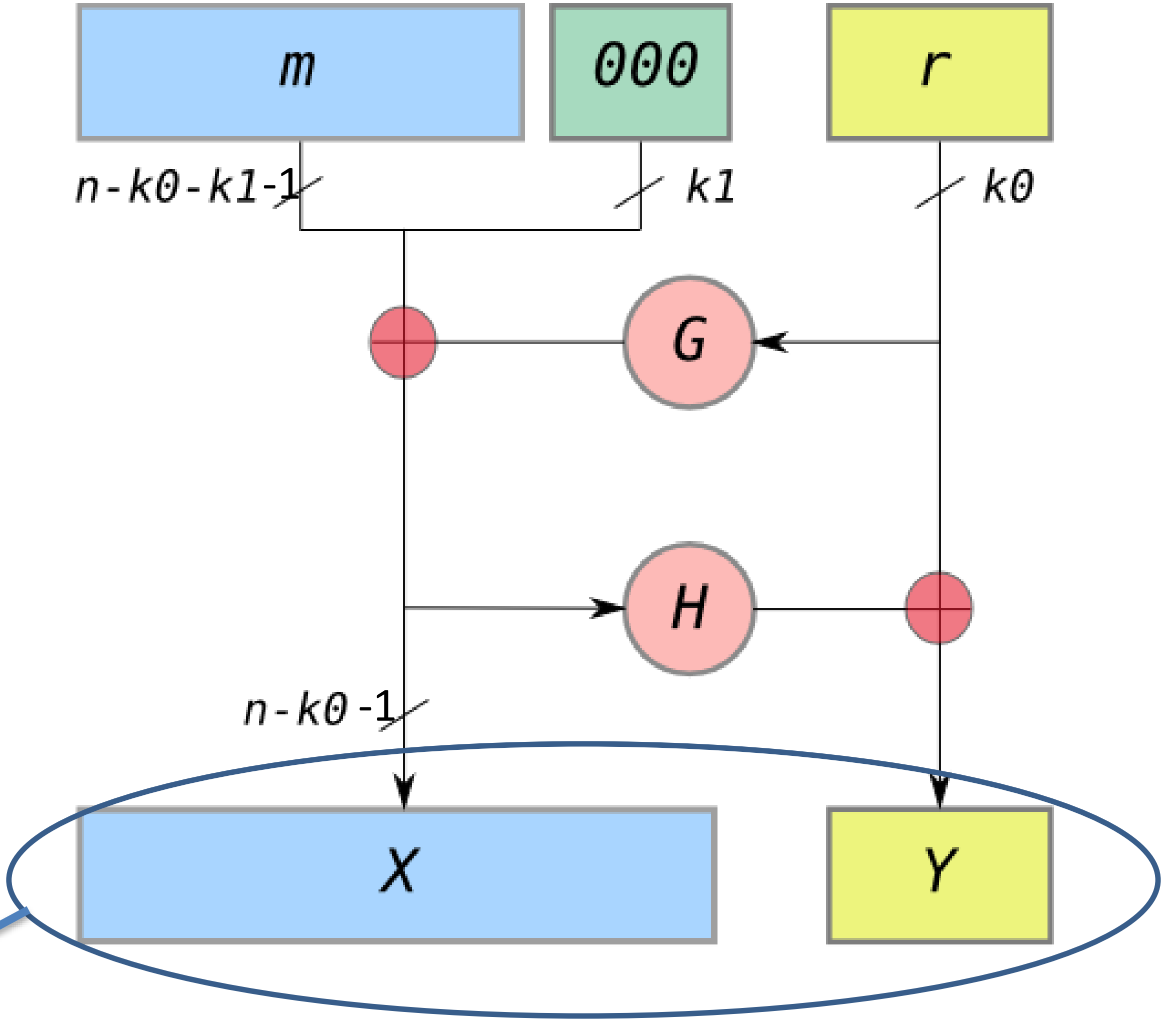
- OAEP is now used for RSA message padding
  - **OAEP**
    - **Optimal asymmetric encryption padding** (1994)
  - Details
    - PKCS#1 v2.1 (the latest version); or RFC 3447

RSA Message Padding: OAEP

- Step1. Generate a one-time secret and random number r for each plaintext
  - r is k_0-bit (at least 128-bit)
- Step 2: Pad the message using 0’s and r to n-1 bits
  - n is the length of RSA modulus
- Step 3: Apply two functions G and H to randomize the padded message
  - Details given in the next slide

RSA Message Padding: OAEP

(the specification here is the simplified version, and is slightly different from RFC)



n: the number of bits in the RSA modulus.

k_0 and k_1: integers fixed by the protocol

m: plaintext message, $(n - k_0 - k_1\text{-}1)$-bit string

G and H: two functions used in the protocol

Then encrypt (x || y)

RSA Message Padding: OAEP

- Security of OAEP
  - "Provably" secure, 1994
    - Complicated security proof
    - Get standardized for its security proof
  - OAEP's security proof was found to be incorrect, 2001
    - But OAEP is still strong enough

RSA Blinding

- RSA blinding is applied to improve the security of RSA decryption against the timing attack
- Timing attack
  – for some cipher, different inputs may result in different processing time
  – the timing attack analyzes the difference in processing time and recovers the secret key

Timing Attack on Modular Exponentiation

Modular Exponentiation:

To compute $y = a^x \bmod n$

$$x = x_{t-1}x_{t-2}\cdots x_2 x_1 x_0$$

The simple algorithm is that

$$y = a,\ z = 1$$

for $i = 0$ to $t-1$ do

{

$\quad$ if $x_i = 1$, then $z = z \cdot y \bmod n$

$\quad$ $y = y^2 \bmod n$

}

Suppose that the attacker knows the overall decryption timing of RSA. How to find the private key?

A simple observation of the square-and-multiply algorithm is that the overall timing is closely related to the number of 1's in the exponent x. The attacker is able to recover this information. But such information is not enough to recover all the bits in x.

Timing Attack on Modular Exponentiation

Modular Exponentiation:

To compute $z = a^x \bmod n$

$$x = x_{t-1}x_{t-2}\cdots x_2 x_1 x_0$$

The simple algorithm is that

$y = a, z = 1$

for $i = 0$ to $t-1$ do

$\{$

$\quad$ if $x_i = 1$, then $z = z \cdot y \bmod n$

$\quad$ $y = y^2 \bmod n$

$\}$

Timing Attack on modular exponentiation:

Knowing $x_0 \cdots x_{i-1}$, to determine x_i

1) find many inputs a' so that at step i,

$\quad z_i = z_{i-1} \times y_{i-1}$ is slow to compute;

2) find many inputs a'' so that

$\quad z_i = z_{i-1} \times y_{i-1}$ is fast to compute

3) Obtain the avarage timing t' of

$\quad$ computing $z = a'^x \bmod n$.

$\quad$ Obtain the average timing t'' of

$\quad$ computing $z = a''^x \bmod n$.

$\quad$ If t' is close to t'', x_i is 0.

$\quad$ If t' is larger than t'', x_i is 1.

RSA Blinding

- Remote practical timing attack on OpenSSL (2003)
  - If the timing variance over the network is less than one millisecond, it is possible to recover the secret key of RSA in OpenSSL with about 1/3 million queries.

RSA Blinding

RSA Blinding is used to defend against the timing attack:

To compute $m = c^d \bmod n$, we generate a one-time random secret number r for each c, then compute

$$t = r^e \bmod n;$$

$$m = ((t \times c)^d \bmod n) \times r^{-1} \bmod n$$

Now the inputs to RSA decryption are random and Secret ➔ not vulnerable to timing attack

RSA Applications

- Used in almost all the secure Internet communication applications
  - Public key infrastructure
  - TLS/SSL
    - Microsoft Outlook, webmail
    - Internet Banking
    - NTULearn
    - ……

Summary

- Public key encryption
  - Allows two parties to communicate secretly without sharing a secret key before communication
- RSA
  - Specifications
  - Implementation
    - Primality testing: Fermat's primality test, Miller-Rabin primality test
    - Extended Euclidean algorithm
    - Fast modular exponentiation
  - Security
    - Integer factorization
      - Dixon's Random Squares algorithm
    - Other attacks
      - Short message
      - Shared public key
      - Small encryption component
      - Small decryption component
  - Secure Implementation
    - p, q, n, d
    - Message padding (OAEP)
      - Never use the "textbook" RSA in practice
      - Message padding is important for the security of RSA
    - RSA blinding