Public-Key Cryptography
RSA
Attacks against RSA

Système et Sécurité

Public Key Cryptography Overview

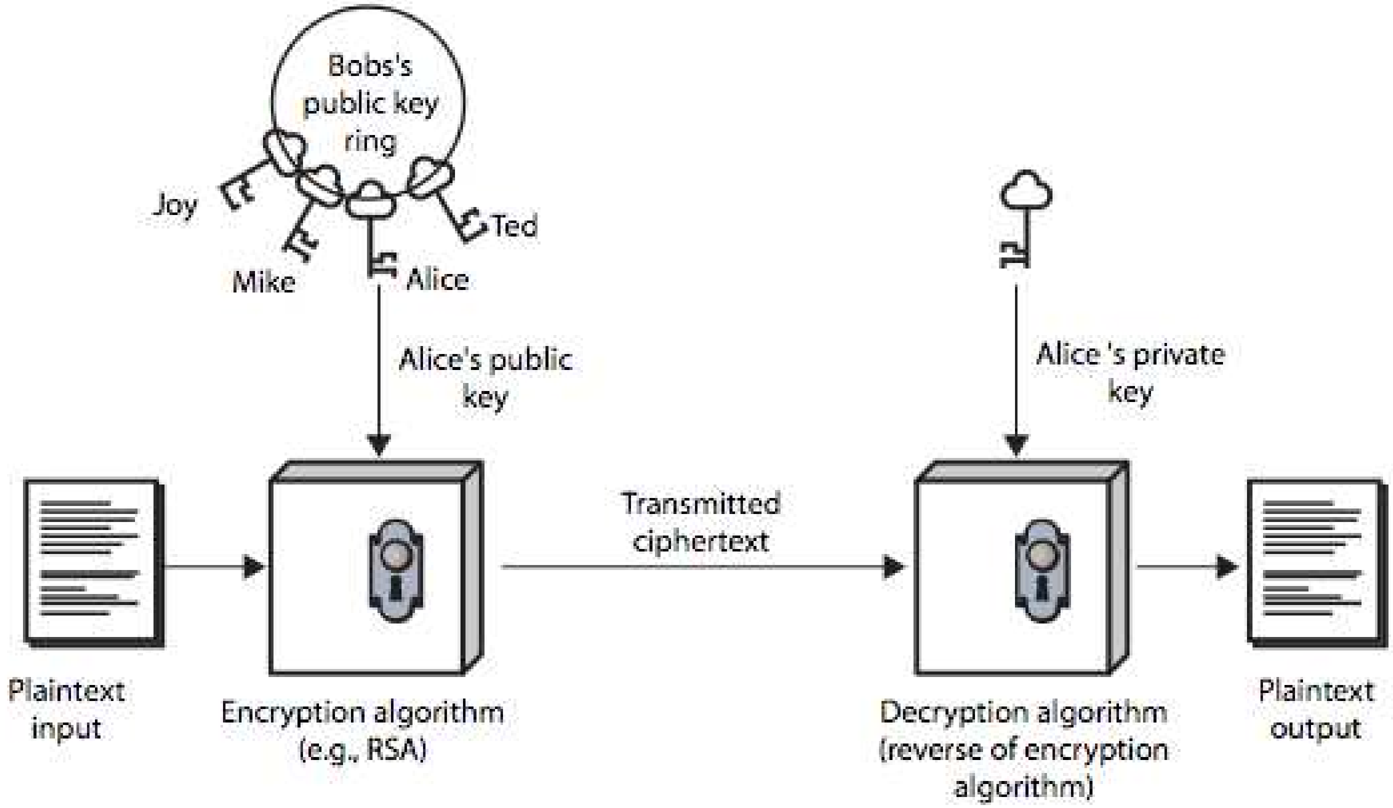
- Proposed in Diffie and Hellman (1976) “New Directions in Cryptography”
  - public-key encryption schemes
  - public key distribution systems
    - Diffie-Hellman key agreement protocol
  - digital signature
- Public-key encryption was proposed in 1970 by James Ellis in a classified paper made public in 1997 by the British Governmental Communications Headquarters
- Diffie-Hellman key agreement and concept of digital signature are still due to Diffie & Hellman

Public Key Encryption

- Public-key encryption
  - each party has a PAIR (K, K^{-1}) of keys: K is the **public** key and K^{-1} is the **private** key, such that

    $\mathbf{D}_{K^{-1}}[\mathbf{E}_K[M]] = M$
- Knowing the public-key and the cipher, it is *computationally infeasible* to compute the private key
- Public-key crypto systems are thus known to be ***asymmetric*** crypto systems
- The public-key K may be made publicly available, e.g., in a publicly available directory
- *Many* can encrypt, *only one* can decrypt

Public-Key Cryptography



Public-Key Encryption Needs One-way Trapdoor Functions

- Given a public-key crypto system,
  - Alice has public key K
  - $\mathbf{E}_K$ must be a one-way function, i.e.: knowing y=$\mathbf{E}_K$ [x], it should be *difficult* to find x
- However, $\mathbf{E}_K$ must **not** be one-way from Alice's perspective. The function $\mathbf{E}_K$ must have a *trapdoor* such that the knowledge of the trapdoor enables Alice to invert it

Trapdoor One-way Functions

- **Definition:**
- A function f: $\{0,1\}^* \rightarrow \{0,1\}^*$ is a trapdoor one-way function iff f(x) is a one-way function; however, given some *extra information* it becomes feasible to compute f^{-1}: given y, find x s.t. y = f(x)

RSA Algorithm

- Invented in **1978** by Ron **R**ivest, Adi **S**hamir and Leonard **A**dleman
  - Published as R. L. Rivest, A. Shamir, L. Adleman, "*On Digital Signatures and Public Key Cryptosystems*", Communications of the ACM, vol. 21 no 2, pp. 120-126, Feb 1978
- Security relies on the difficulty of *factoring large composite numbers*
- Essentially the same algorithm was discovered in 1973 by Clifford Cocks, who works for the British intelligence



Z_{pq}*

- Let p and q be two large primes
- Denote their product n=pq.
- $Z_n^* = Z_{pq}^*$ contains, by definition, all integers in the range [1,pq-1] that are relatively prime to both p and q
- The size of Z_n^* is

$$\Phi(pq) = (p-1)(q-1)=n-(p+q)+1$$

- For every $x \in Z_{pq}^*$, $x^{(p-1)(q-1)} \equiv 1 \bmod n$

Exponentiation in Z_{pq}*

- Motivation: We want to use exponentiation for encryption

- Let e be an integer, $1<e<(p-1)(q-1)$
- When is the function $f(x)=x^e$ a *one-to-one* function in Z_{pq}*?
- If x^e is one-to-one, then it is a *permutation* in Z_{pq}*

Exponentiation in Z_{pq}*

- Claim: If e is ***relatively prime*** to (p-1)(q-1) then $f(x)= x^e$ is a one-to-one function in Z_{pq}*

- *Proof* by constructing the inverse function of f()
  As gcd{e,(p-1)(q-1)}=1, then there exists d and k s.t. → ed=1+k(p-1)(q-1)
- Let $y= x^e$, then $y^d=(x^e)^d=x^{1+k(p-1)(q-1)}=x \pmod{pq}$, i.e., $g(y)= y^d$ is the inverse of $f(x)= x^e$.

RSA Public Key Crypto System

- **Key generation:**
  - Select 2 large prime numbers of about the same size, p and q
  - Compute $n = pq$, and $\Phi(n) = (p-1)(q-1)$
  - Select a random integer e, $1 < e < \Phi(n)$, s.t. $\gcd(e, \Phi(n)) = 1$
  - Compute d, $1< d < \Phi(n)$ s.t. $ed \equiv 1 \bmod \Phi(n)$ (using the Extended Euclidean Algorithm)
- **Public key: (e, n)**
- **Private key: d**
- **<u>Note</u>: p and q must remain secret**

RSA Description (cont.)

- **Encryption**
  - Given a message M, $0 < M < n$ $M \in Z_n - \{0\}$
  - use public key (e, n)
  - compute $C = M^e \bmod n$ $C \in Z_n - \{0\}$
- **Decryption**
  - Given a ciphertext C, use private key (d)
  - Compute $C^d \bmod n = (M^e \bmod n)^d \bmod n =$
    $= M^{ed} \bmod n = M$

RSA Example (1)

- p = 17, q = 11, n = 187, Φ(n) = 160
- Let us choose e=7, since gcd (7,160)=1
- Let us compute d: de=1 mod 160 , d=23 (in fact, 23x7=161 = 1 mod 160

- Public key = {7,187}
- Secret key = 23

RSA Example (1) cont.

- Given message (plaintext) M= 88
  (note that 88<187)

- Encryption:
  $C = 88^7 \bmod 187 = 11$
- Decryption:
  $M = 11^{23} \bmod 187 = 88$

RSA Example (2)

- $p = 11, q = 7, n = 77, \Phi(n) = 60$
- $e = 37, d = 13$ ($ed = 481$; $ed \bmod 60 = 1$)

- Let $M = 15$. Then $C \equiv M^e \bmod n$

  $C \equiv 15^{37} \pmod{77} = 71$
- $M \equiv C^d \bmod n$

  $M \equiv 71^{13} \pmod{77} = 15$

Why does RSA work?

- Need to show that $(M^e)^d \pmod{n} = M$, $n = pq$

- Since $ed \equiv 1 \pmod{\Phi(n)}$
  We have that $ed = t\Phi(n) + 1$, for some integer t.

- So:
  $(M^e)^d \pmod{n} = M^{t\Phi(n) + 1} \pmod{n} =$
  $(M^{\Phi(n)})^t M^1 \pmod{n} = 1^t M \pmod{n} = M \pmod{n}$
  as desired.

RSA Implementation

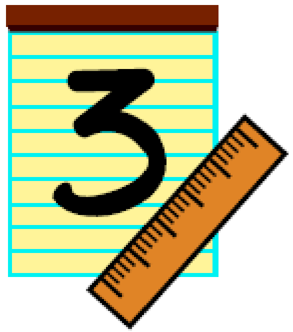
- n, p, q
- The security of RSA depends on how large n is, which is often measured in the number of bits for n. Current recommendation is 1024 bits for n.
- p and q should have the same bit length, so for 1024 bits RSA, p and q should be about 512 bits.
- … but *p-q* should <u>*not*</u> be small!

RSA Implementation

- Select p and q prime numbers
- In practice, select random numbers, then test for primality
- Many implementations use the Rabin-Miller test, (probabilistic test)

RSA Implementation

- e
- e is usually chosen to be 3 or $2^{16} + 1 = 65537$
  - Binary: 11 or 10000000000000001
- In order to speed up the encryption
- the smaller the number of 1 bits, the better
- why?



Square-and-Multiply Algorithm for Modular Exponentiation

- Modular exponentation means “Computing x^c mod n”
- In RSA, both encryption and decryption are modular exponentations.
- Obviously, the computation of x^c mod n can be done using c-1 modular multiplication, but this is very inefficient if c is large.
- Note that in RSA, c can be as big as $\Phi(n) - 1$.

- The well-known “square-and-multiply” approach reduces the number of modular multiplications required to compute x^c mod n to at most 2k, where k is the number of bits in the *binary representation* of c.

Square-and-Multiply Algorithm for Modular Exponentiation

- "Square-and-multiply" assumes that the exponent c is represented in binary notation, say :

$$c = \sum_{i=0}^{k-1} c_i \, 2^i$$

Algorithm: Square-and-multiply (x, n, c = c_{k-1} c_{k-2} … c_1 c_0)

```
z=1
for i = k-1 downto 0 {
        z = z^2 mod n
        if c_i = 1 then z = (z * x) mod n
}
return z
```

Square-and-Multiply Algorithm for Modular Exponentiation: Example

- Let us compute 9726^{3533} mod 11413
- x=9726, n=11413, c=3533 = 110111001101 (binary form)

i	c_i	***z***
11	1	1^2x 9726=9726
10	1	9726^2x 9726=2659
9	0	2659^2=5634
8	1	5634^2x 9726=9167
7	1	9167^2x 9726=4958
6	1	4958^2x 9726=7783
5	0	7783^2=6298
4	0	6298^2=4629
3	1	4629^2x 9726=10185
2	1	10185^2x 9726=105
1	0	105^2=11025
0	1	11025^2x 9726=**5761**

Probabilistic Primality Testing

- In setting up the RSA Cryptosystem, it is necessary to generate large « random primes ».
- In practice this is done by generating large random numbers and then test them for primality using a *probabilistic polynomial-time* Montecarlo algorithm like Solovay-Strassen or Miller-Rabin algorithm.
- Both these algorithms are fast: an integer n can be tested in time that is polynomial in $\log_2 n$, the number of bits in the binary representation of n
- However, there is a possibility that the algorithm claims that n is prime when it is **not**
- Running the algorithm enough times, one can reduce the error probability <u>below any desired threshold</u>.

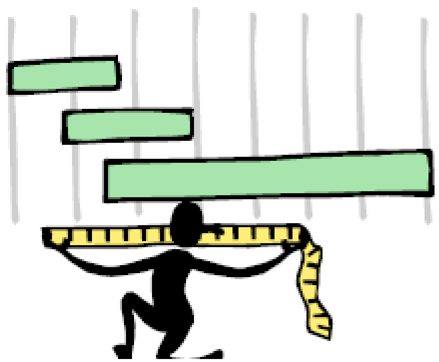
Probabilistic Primality Testing

- How many random integers (of a specifiz size, say 500 bits) will need to be tested until we find one that is prime?

- The Prime Number Theorem states that the number of primes not exceeding N tends to N/ln N, for large N values.

RSA on Long Messages

- RSA requires that the message M is at most n-1 where n is the size of the modulus.
- What about longer messages?
  - They are broken into blocks.
  - Smaller messages are padded.
  - CBC is used to prevent attacks regarding the blocks.



- NOTE: In practice RSA is used to encrypt symmetric keys, so the message is not very long.

Digital Signature

- The fact that the encryption and decryption operations are inverses and operate on the same set of inputs also means that the operations can be employed *in reverse order* to obtain a digital signature scheme following Diffie and Hellman's model.
- A message M can be digitally signed by applying the *decryption* operation to it, i.e., by exponentiating it to the d^{th} power
  - $s = \text{SIGN}(M) = M^d \bmod n$

Digital Signature

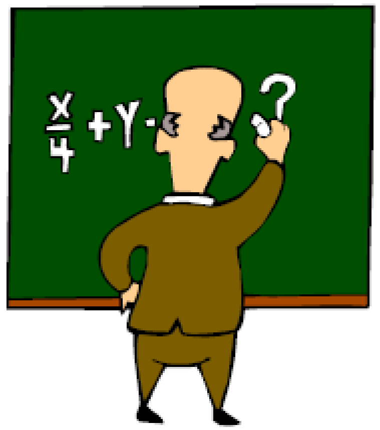
- The digital signature can then be verified by applying the *encryption* operation to it and comparing the result with and/or recovering the message:
  - $M = VERIFY(s) = s^e \bmod n$

- In practice, the plaintext M is generally some function of the message, for instance a formatted one-way hash of the message.
- This makes it possible to sign a message of any length with only one exponentiation.

Attacks against RSA

Math-Based Key Recovery Attacks

- Three possible approaches:
  1. Factor n = pq
  2. Determine Φ(n)
  3. Find the private key d directly
- All the above are equivalent to factoring n



Knowing $\Phi(n)$ Implies Factorization

- If a cryptanalyst can learn the value of $\Phi(n)$, then he can factor n and break the system. In other words, computing $\Phi(n)$ is no easier than factoring n
- In fact, knowing both n and $\Phi(n)$, one knows

$$n = pq$$

$$\Phi(n) = (p-1)(q-1) = pq - p - q + 1 = n - p - n/p + 1$$

$$p\Phi(n) = np - p^2 - n + p$$

$$p^2 - np + \Phi(n)p - p + n = 0$$

$$p^2 - (n - \Phi(n) + 1)\,p + n = 0$$

- There are two solutions of p in the above equation.
- Both p and q are solutions.

Knowing Φ(n) Implies Factorization

- Example: suppose the cryptalyst has learned that n = 84773093 and Φ(n)=84754668.
- Find out the two factors of n.

Knowing Φ(n) Implies Factorization

- Example: suppose the cryptalyst has learned that n = 84773093 and Φ(n)=84754668.
- Find out the two factors of n.

- Equation: p^2 -18426p+84773093=0
- Solutions: 9539 and 8887

Factoring Large Numbers

- **RSA-640 bits, Factored Nov. 2 2005**
- **RSA-200 (663 bits) factored in May 2005**
- **RSA-768 has 232 decimal digits and was factored on December 12, 2009, latest.**
- Three most effective algorithms are
  - quadratic sieve
  - elliptic curve factoring algorithm
  - number field sieve

Decryption attacks on RSA

- **RSA Problem:** Given a positive integer n that is a product of two distinct large primes p and q, a positive integer e such that gcd(e, (p-1)(q-1))=1, and an integer c, find an integer m such that $m^e \equiv c \pmod{n}$
  - widely believed that the RSA problem is computationally equivalent to integer factorization; however, no proof is known

- **The security of RSA encryption's scheme depends on the hardness of the RSA problem.**

Summary of Key Recovery Math-based Attacks on RSA

- Three possible approaches:
  1. Factor n = pq
  2. Determine $\Phi(n)$
  3. Find the private key d directly
- All are equivalent
  - finding out d implies factoring n
  - if factoring is hard, so is finding out d

Finding d: Timing Attacks

- *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems (1996), Paul C. Kocher*
- By measuring the time required to perform decryption (exponentiation with the private key as exponent), an attacker can figure out the private key
- Possible countermeasures:
  – use constant exponentiation time
  – add random delays
  – blind values used in calculations

Timing Attacks (cont.)

- Is it possible in practice? YES !

OpenSSL Security Advisory [17 March 2003]
Timing-based attacks on RSA keys
==================================
OpenSSL v0.9.7a and 0.9.6i vulnerability

- Researchers have discovered a timing attack on RSA keys, to which OpenSSL is generally vulnerable, unless RSA blinding has been turned on.
- RSA blinding: the decryption time is no longer correlated to the value of the input ciphertext
- Instead of computing c^d mod n, choose a secret random value r and compute $(r^e c)^d$ mod n.
- A new value of r is chosen for each ciphertext