The RSA cryptosystem

Public Key Encryption

RSA in a nutshell

- Rivest-Shamir-Adleman, 1978
  - **Rivest**, R.; **Shamir**, A.; **Adleman**, L. (February 1978). "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," Communications of the ACM 21 (2): 120–126. doi: 10.1145/359340.359342.
- The most widely used asymmetric crypto-system
- Many applications
  - Encryption of small pieces (e.g. key transport)
  - Digital Signatures
- Underlying one-way function: integer factorization problem

RSA key generation

1. Generate two large, distinct primes ***p, q*** (100÷200 decimal digits)
2. Compute $n = p \times q$ and $\varphi(n) = (p-1)\times(q-1)$
3. Select a random number $1 < e < \varphi(n)$ such that $\gcd(e, \varphi(n)) = 1$
4. Compute the unique integer $1 < d < \varphi$ such that $ed \equiv 1 \pmod{\varphi}$
5. **(*d*, *n*)** is the *private* key
6. **(*e*, *n*)** is the *public* key

At the end of key generation, ***p*** and ***q*** must be destroyed

RSA encryption and decryption

Encryption*.* To generate *c* from *m*, Bob should do the following

1. Obtain *A*'s *authentic* public key (n, e)
2. Represent the message as an integer *m* in the interval [0, *n*-1]
3. Compute $c = m^e \bmod n$
4. Send *c* to *A*

Decryption. To recover *m* from *c*, Alice should do the following

1. Use the private key d to recover $m = c^d \bmod n$

RSA consistency

- We have to prove that ***D*(*d*(*E*(e, *m*)) = *m***, i.e., $c^d \equiv m \pmod{n}$
- The proof may be based on either the **Fermat's little theorem** or the **Eulero's theorem**

RSA consistency

Proof based on Fermat's little theorem

- **Fermat's little theorem**
  - If p is prime and gcd(p, a) = 1, then $a^{p-1} = 1 \pmod{p}$
- **Proof**
  - Since ed = 1 mod φ then ed = 1 + t (p – 1)(q – 1)
  - Check whether x = y mod (pq) is equivalent to check whether x = y (mod p) ∧ x = y (mod q)
  - $m^{ed} = m \pmod{p}$
    - m = 0 (mod p), so m is a multiple of p so $m^{ed} = 0 = m \pmod{p}$
    - $m \neq 0 \pmod{p}$, $m^{ed} = m\, m^{t(p-1)(q-1)} = m\, (m^{(p-1)})^{t(q-1)} = m\, (1)^{t(q-1)} = m \pmod{p}$
  - Proof for q proceeds in a similar way

RSA consistency
Proof based on Eulero's theorem

UNIVERSITÀ DI PISA

- **Eulero's theorem**
  - **∀integer $n > 1$, ∀$a \in \mathbb{Z}_n^*$, $a^{\varphi(n)} \equiv 1 \pmod{n}$** *where* **$\mathbb{Z}_n^* = \{ x \mid 1 < x < n, \gcd(x, n) = 1\}$**
- **Proof**
  - We have to prove that ***D(d(E(e, m)) = m***, i.e., **$c^d \equiv m^{de} \equiv m^{t\cdot\varphi(n)+1} \pmod{n}$**, where ***t*** is some integer ⇒ **$m^{t\cdot\varphi(n)} \cdot m^1 \equiv (m^{\varphi(n)})^t \cdot m^1 \equiv m \pmod{n}$**

Example with artificially small numbers

UNIVERSITÀ DI PISA

Key generation

- Let $p = 47$ e $q = 71$
  $n = p \times q = 3337$
  $\varphi = (p-1) \times (q-1) = 46 \times 70 = 3220$
- Let $e = 79$
  $ed = 1 \bmod \varphi$
  $79 \times d = 1 \bmod 3220$
  $d = 1019$

Encryption

Let $m = 9666683$

Divide m into blocks $m_i < n$

$m_1 = 966$; $m_2 = 668$; $m_3 = 3$

Compute

$c_1 = 966^{79} \bmod 3337 = 2276$

$c_2 = 668^{79} \bmod 3337 = 2423$

$c_3 = 3^{79} \bmod 3337 = 158$

$c = c_1c_2c_3 = 2276\ 2423\ 158$

Decryption

$m_1 = 2276^{1019} \bmod 3337 = 966$

$m_2 = 2423^{1019} \bmod 3337 = 668$

$m_3 = 158^{1019} \bmod 3337 = 3$

$m = 966\ 668\ 3$

RSA

- RSA algorithms for key generation, encryption and decryption are "easy"
- They involve the following operations
  - Discrete exponentiation
  - Generation of large primes (see next slide)
  - Solving diophantine equations

How to find a large prime

repeat
    $p \leftarrow$ randomOdd(x);
until isPrime(p);

- **FACT**. On average (ln x)/2 odd numbers must be tested before a prime $p < x$ can be found

- Primality tests **do not** try to factor the number under test
  - *probabilistic primality test* (Solovay-Strassen, Miller-Rabin) polynomial in **log n**
  - *true primality test* (O(n^{12}) in 2002))

On computing the private exponent *d*

- Solution of **$d \cdot e \equiv 1 \bmod \varphi(n)$** with **$gcd(e, \varphi(n)) \equiv 1$** can be done by means of the **Extended Euclidean Algorithm** (EEA)
  - Exponent ***d*** can be computed efficiently (polytime)
  - Condition $gcd(e, \varphi(n)) \equiv 1$

Modular ops - complexity

Bit complexity of basic operations in Z_n

- Let **n** be on **k** bits (**$n < 2^k$**)
- Let **a** and **b** be two integers in **Z_n** (on k-bits)
  - **Addition $a + b$** can be done in time **$O(k)$**
  - **Subtraction $a – b$** can be can be done in time **$O(k)$**
  - **Multiplication $a \times b$** can be done in **$O(k^2)$**
  - **Division $a = q \times b + r$** can be done in time **$O(k^2)$**
  - **Inverse a^{-1}** can be done in **$O(k^2)$**
  - **Modular exponentiation a^k** can be done in **$O(k^3)$**

How to encrypt/decrypt efficiently

- RSA requires *modular exponentiation* c^d **mod** ***n***
  - Let ***n*** have ***k*** bits in its binary representation, ***k = log n + 1***
- ***Grade-school*** algorithm requires ***(d-1)*** modular multiplications
  - ***d*** is as large as **n** which is exponentially large with respect to ***k***
  - The grade-school algorithm is inefficient
- ***Square-and-multiply*** algorithm requires up to **2k** multiplications thus the algorithm can be done in **O(*k*^{3})**

How to encrypt/decrypt efficiently

- RSA requires *modular exponentiation* a^x **mod** ***n***
  - Let ***n*** have ***k*** bits in its binary representation, ***k = log n + 1***
- ***Grade-school*** algorithm requires ***(x-1)*** modular multiplications
  - If ***x*** is as large as **n,** which is exponentially large with respect to ***k*** ➔ the grade-school algorithm is inefficient
- ***Square-and-multiply*** algorithm requires up to **2k** multiplications thus the algorithm can be done in **O(*k*^{3})**

How to encrypt and decrypt efficiently

UNIVERSITÀ DI PISA

Exponentiation by repeated squaring and multiplication: m^e **mod** n requires **at most** $\log_2(e)$ multiplications and $\log_2(e)$ squares

Let $e_{k-1}, e_{k-2}, \ldots, e_2, e_1, e_0$, where $k = \log_2 e$, the binary representation of **e**

$$m^e \bmod n = m^{\left(e_{k-1}2^{k-1}+e_{k-2}2^{k-2}+\cdots+e_2 2^2+e_1 2+e_0\right)} \bmod n \equiv$$
$$m^{e_{k-1}2^{k-1}} m^{e_{k-2}2^{k-2}} \cdots m^{e_2 2^2} m^{e_1 2} m^{e_0} \bmod n \equiv$$
$$\left(m^{e_{k-1}2^{k-2}} m^{e_{k-2}2^{k-3}} \cdots m^{e_2 2} m^{e_1}\right)^2 m^{e_0} \bmod n \equiv$$
$$\left(\left(m^{e_{k-1}2^{k-3}} m^{e_{k-2}2^{k-4}} \cdots m^{e_2}\right)^2 m^{e_1}\right)^2 m^{e_0} \bmod n \equiv$$
$$\left(\left(\left(\left(m^{e_{k-1}}\right)^2 m^{e_{k-2}}\right)^2 \cdots m^{e_2}\right)^2 m^{e_1}\right)^2 m^{e_0} \bmod n$$

```
c ← 1
for (i = k-1; i >= 0; i --) {
    c ← c² mod n;
    if (e_i == 1)
        c ← c × m mod n;
}
```

- always k square operations
- **at most k** modular multiplications (***equal to the number of 1 in the binary representation of e***)

Square and multiply

UNIVERSITÀ DI PISA

Exponentiation by repeated squaring and multiplication: a^x **mod** n requires **at most** $\log_2(x)$ multiplications and $\log_2(x)$ squares

Let $x_{k-1}, x_{k-2}, \ldots, x_2, x_1, x_0$, where $k = \log_2 x$, the binary representation of x

$$a^x \bmod n = a^{\left(x_{k-1}2^{k-1}+x_{k-2}2^{k-2}+\cdots+x_2 2^2+x_1 2+x_0\right)} \bmod n \equiv$$
$$a^{x_{k-1}2^{k-1}} a^{x_{k-2}2^{k-2}} \cdots a^{x_2 2^2} a^{x_1 2} a^{x_0} \bmod n \equiv$$
$$\left(a^{x_{k-1}2^{k-2}} a^{x_{k-2}2^{k-3}} \cdots a^{x_2 2} a^{x_1}\right)^2 a^{x_0} \bmod n \equiv$$
$$\left(\left(a^{x_{k-1}2^{k-3}} a^{x_{k-2}2^{k-4}} \cdots a^{x_2}\right)^2 a^{x_1}\right)^2 a^{x_0} \bmod n \equiv$$

...

$$\left(\left(\left(\left(a^{x_{k-1}}\right)^2 a^{x_{k-2}}\right)^2 \cdots a^{x_2}\right)^2 a^{x_1}\right)^2 a^{x_0} \bmod n$$

```
c ← 1
for (i = k-1; i >= 0; i --) {
    c ← c² mod n;
    if (x_i == 1)
        c ← c × a mod n;
}
```

- always **k square operations**
- **at most k modular multiplications** (*equal to the number of 1 in the binary representation of e*)

Fast encryption with short public exponent

- RSA ops with public key exponent ***e*** can be speeded-up
  - Encryption
  - Digital signature verification
- The public key ***e*** can be chosen to be a very small value
  - e = 3 #MUL + #SQ = 2
  - e = 17 #MUL + #SQ = 5
  - $e = 2^{16}+1$ #MUL + #SQ = 17
  - **RSA is still secure**
- There is no easy way to accelerate RSA when the private exponent ***d*** is involved
  - Len d = len n

RSA one-way function

- One-way function y = f(x)
  - y = f(x) is easy
  - $x = f^{-1}(y)$ is hard
- RSA one-way function
  - Multiplication is easy
  - Factoring is hard

Security of RSA

The RSA Problem (RSAP)

- **DEFINITION. The RSA Problem** (**RSAP**): recovering plaintext *m* from ciphertext *c*, given the public key (*n*, *e*)

RSA VS FACTORING

- **FACT. RSAP $\leq_P$ FACTORING**
  - FACTORING is at least as difficult as RSAP or, equivalently, RSAP is not harder than FACTORING
  - ***It is widely believed that RSAP and Factoring are computationally equivalent, although no proof of this is known.***

Security of RSA

- **THM (FACT 1)**. Computing the decryption exponent ***d*** from the public key (***n, e***) is computationally equivalent to factoring ***n***
  a. If the adversary could somehow factor ***n***, then he could subsequently compute the private key ***d*** efficiently
  b. If the adversary could somehow compute ***d***, then it could subsequently factor ***n*** efficiently

Security of RSA

RSAP and e-th root

- A possible way to decrypt $c = m^e \bmod n$ is to compute the *modular* *e*-th root of *c*
- **THM (FACT 2)**. Computing the *e*-th root is a computationally easy problem iff *n* is prime
- **THM (FACT 3)**. If *n* is composite the problem of computing the *e*-th root is *equivalent* to factoring

Security of RSA

- **THM (FACT 4)**. Knowing φ is computationally equivalent to factoring
- **PROOF**.

1. **Given p and q**, s.t. $n = pq$, computing φ is immediate.
2. **Let φ be given**.
   a. From $\varphi = (p-1)(q-1) = n - (p+q) + 1$, determine $x_1 = (p+q)$.
   b. From $(p - q)^2 = (p + q)^2 - 4n = x_1^2 - 4n$, determine $x_2 = (p - q)$.
   c. Finally, $p = (x1 + x2)/2$ and $q = (x1 - x2)/2$.

Security of RSA

UNIVERSITÀ DI PISA

- Exhaustive Private Key Search
  - This attack could be more difficult than factoring *d*
  - Key d is the same order of magnitude as n thus it is much greater than p and q

Factoring

UNIVERSITÀ DI PISA

- **Primality testing vs. factoring**
  - (**FACT 5**) Deciding whether an integer is composite or prime seems to be, in general, much easier than the factoring problem
- **Factoring algorithms**
  - Brute force
  - Special purpose
  - General purpose
  - Elliptic Curve
  - Factoring on Quantum Computer (for the moment only theorethical)

Factoring algorithms

UNIVERSITÀ DI PISA

- **Brute Force**
  - Unfeasible if *n* large and *p* len = *q* len
- **General purpose**
  - The running time depends solely on the size of n
    - Quadratic sieve
    - General number field sieve
- **Special purpose**
  - The running time depends on certain properties
    - Trial division
    - Pollard's rho algorithm
    - Pollard's *p* -1 algorithm
- **Elliptic curve algorithm**

Factoring: running times

UNIVERSITÀ DI PISA

Trial division: $O\left(\sqrt{n}\right)$

Quadratic sieve: $O\left(e^{\left(\sqrt{\ln(n)\bullet\ln\ln(n)}\right)}\right)$

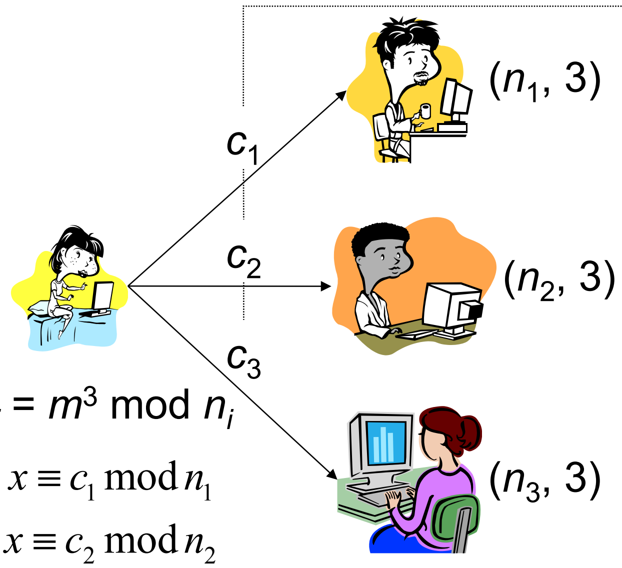
General number field sieve: $O\left(e^{\left(1.923\times\sqrt[3]{\ln(n)\bullet(\ln\ln(n))^2}\right)}\right)$

RSA in practice

Selecting primes p and q

- ***p*** and ***q*** should be selected so that factoring ***n* = *pq*** is computationally infeasible, therefore
- ***p*** and ***q*** should be ***sufficiently large*** and about the ***same bitlenght*** (to avoid the elliptic curve factoring algorithm)
- ***p - q should be not too small***

RSA: low exponent attack



$c_i = m^3 \bmod n_i$

$$\begin{cases} x \equiv c_1 \bmod n_1 \\ x \equiv c_2 \bmod n_2 \\ x \equiv c_3 \bmod n_3 \end{cases}$$

- If n_1, n_2 ed n_3 are pairwise coprime, use CRT to find x = $m^3 \bmod n_1 n_2 n_3$
- *As* $m < n_i$ by RSA encryption definition then $m^3 < n_1 n_2 n_3$, then $x = m^3$
- Thus an eavesdropper recovers m by computing the *integer cube root* of x (non modular!)

RSA in practice - padding

- We have described schoolbook/plain RSA
- Plain RSA implementation may be insecure
  - RSA is deterministic
  - PT values x = 0, x = 1 produce CT equal to 0 and 1
  - Small PT might be subject to attacks
  - RSA is malleable
- **Never use plain RSA**
- Padding is a possible solution
  - Optimal Asymmetric Encryption Padding (OAEP) in Public Key Cryptography Standard #1 (PKCS #1)

RSA is malleable

- RSA malleability is based on the **homo-morphic property** of RSA
- Attack
  - The attacker replaces CT = y mod n by CT' = $s^e \bullet y$ mod n, with s some integer s.t. gcd(s, n) = 1
  - The receiver decrypts CT': $(s^e \bullet y)^d = s^{ed} \bullet x^{ed} = s \bullet x \bmod n$
  - By operating on the CT the adversary manages to multiply PT by s
  - **EX**. Let x be an amount of money. If s = 2 then the adversary doubles the amount
  - **Possible solution**: introduce redundancy: ex. $x \,||\, x$

RSA – Homomorphic property

UNIVERSITÀ DI PISA

- Let m_1 and m_2 two plaintext messages
- Let c_1 and c_2 their respective encryptions
- Observe that

$$(m_1 m_2)^e \equiv m_1^e m_2^e \equiv c_1 c_2 \pmod{n}$$

- In other words, the CT of the product $m_1 m_2$ is the product of CTs $c_1 c_2$ mod n

RSA in practice - PKCS #1

UNIVERSITÀ DI PISA

- Parameters
  - M = message
  - | M | = message len in bytes
  - k = | n | modulus len in bytes
  - | H | = hash function output len in bytes
  - L = optional label ("" by default)

RSA in practice - PKCS #1

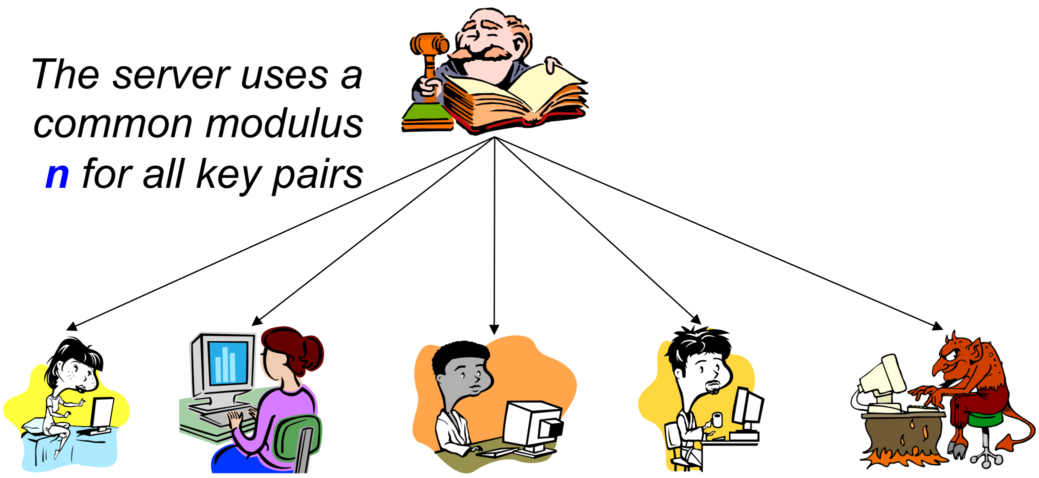
- **Padding**
  1. Generate a string *PS* = 00…0; *PS* len = $k - |M| - 2|H| - 2$ (*PS* len may be zero)
  2. *DB* = Hash(*L*) || *PS* || `0x01` || *M*
  3. *seed* = random(); *seed* len = | H |
  4. *dbMask* = MGF (*seed*, $k - |H| - 1$) (*)
  5. *maskedDB* = *DB* **xor** *dbMask*
  6. *seedMask* = MGF(*maskedDB*, | *H* |)
  7. *maskedSeed* = *seed* **xor** *seedMask*
  8. *EM* = `0x00` || *maskedSeed* || *maskedDB* (**)

(*) *MGF mask generation function (e.g., SHA-1)*

(**) *EM is the padded message*

Common modulus attack

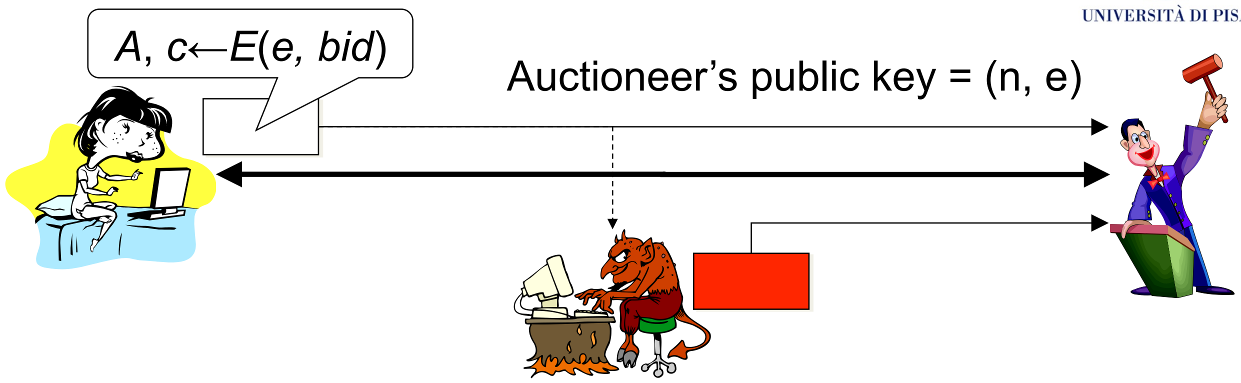
The server uses a common modulus ***n*** *for all key pairs*



(n, e_1) (n, e_2) (n, e_3) (n, e_4) (n, e_5)

- Mr Lou Cipher can efficiently factor *n* from d_5 (FACT 1) and then
- compute all *d*'s

Chosen-plaintext attack

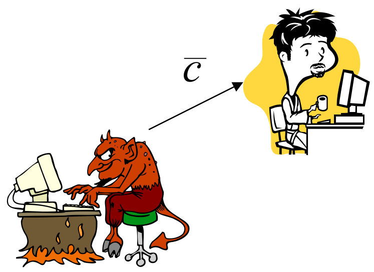


The adversary encrypts all possible bids (e.g, 2^{32}) until he finds a **b** such that *E*(e, b) = *c*

Thus, the adversary sends a bid containing the minimal offer to win the auction: b' = b + 1

Salting is a solution: *r* ← random(); *c*←*E*(*e, r* || *bid*)

An adaptive chosen-ciphertext attack



- Bob decrypts ciphertext except a given ciphertext *c*
- Mr Lou Cipher wants to determine the ciphertext corresponding to *c*

- Mr Lou Cipher selects *x* at random, s.t. gcd(*x*, *n*) =1, and sends Bob the quantity $\bar{c} = cx^e \bmod n$
- Bob decrypts it, producing $\bar{m} = (\bar{c})^d = c^d x^{ed} = mx \pmod n$
- Mr Lou Cipher determine *m* by computing $m = \bar{m}x^{-1} \bmod n$

The attack can be contrasted by imposing structural constraints on *m*