Algoritmi di Crittografia (2023/24)
Notebook 5

In [ ]:
```
from IPython.display import HTML
HTML('<style>{}</style>'.format(open('/home/mauro/.jupyter/custom/custom.css').read()))
```

Latex definitions

In [ ]:
```
# Import delle funzioni/moduli utilizzati nel notebook
from Crypto.PublicKey import RSA
from Crypto.Random.random import randint
from Crypto.Random import get_random_bytes
from Crypto.Util import number
from Crypto.Cipher import AES, PKCS1_OAEP
from math import sqrt
```

Il sistema crittografico a chiave pubblica RSA

- Il sistema crittografico RSA prende il nome dalle iniziali dei loro inventori, Ronald L. Rivest, Adi Shamir e Leonard M. Adlemam, che nel 2002 hanno ricevuto il Premio Turing: for their ingenious contribution to making public-key cryptography useful in practice, di fatto proprio per l'invenzione dell'RSA.
- RSA basa la sua efficacia, in termini di sicurezza, sulla difficoltà computazionale del problema della fattorizzazione di numeri interi, ovvero sul fatto che ad oggi non esistono algoritmi efficienti per fattorizzare numeri interi di grandi dimensioni.
- In realtà, ad oggi, non esiste argomentazione matematica che dimostri che saper violare efficientemente RSA implichi saper anche efficientemente fattorizzare numeri interi
- Non è tuttavia nemmeno noto un procedimento alternativo alla fattorizzazione per violare il codice e si ritiene che non ci sia
- Un sistema crittografico equivalente alla fattorizzazione è stato invece proposto (solo poco dopo la pubblicazione dell'RSA) da un altro premio Turing, l'israeliano, Michael O. Rabin; lo vedremo dopo aver analizzato RSA
- Come ultima nota generale, ricordiamo che un computer quantistico sarebbe in grado di fattorizzare efficientemente i numeri interi e dunque anche violare un sistema crittografico basato sull'algoritmo RSA

Il protocollo base (Textbook RSA)

- Descriviamo ora l'algoritmo base e la "matematica" che ci sta dietro
- Vedremo esempi di generazione delle chiavi e di uso del protocollo di cifratura.
- Analizzeremo poi alcune potenziali debolezze e vedremo (a grandi linee) come viene inserito in un protocollo reale in grado di fornire garanzie di sicurezza molto maggiori rispetto al protocollo base.

L'algoritmo in dettaglio

- Come in tutti i protocolli asimmetrici, la generazione delle chiavi è eseguita dal destinatario dei messaggi cifrati, che supporremo sia Alice.
- Il protocollo ha un solo parametro di input, che è la lunghezza in bit, indicata con N, dell'intero con cui si effettueranno le riduzioni modulari.
- Vediamo i dettagli dell'algoritmo.

Generazione delle chiavi (Alice)

1. genera due numeri primi a caso, p e q, di lunghezza $N/2$ bit;
2. calcola il prodotto $n = p \cdot q$;
3. calcola la quantità $\phi(n) = (p-1) \cdot (q-1)$;
4. determina (in modo sostanzialmente arbitrario), un intero e relativamente primo con $\phi(n)$, tale cioè che $\mathrm{MCD}(e, \phi(n)) = 1$;
5. calcola l'inverso moltiplicativo d di e modulo $\phi(n)$, che esiste proprio perché e è relativamente primo con $\phi(n)$;
6. diffonde la coppia (e, n) come chiave pubblica e conserva d come chiave segreta.

Cifratura di un messaggio M (Bob)

1. si procura la chiave pubblica (e, n) di Alice;
2. calcola $C = M^e \bmod n$;
3. invia C ad Alice come messaggio cifrato

Decifrazione del messaggio C (Alice)

1. calcola $M = C^d \bmod n$

- Prima di dimostrare la correttezza del protocollo, vediamo qualche semplice esempio

Qualche esempio illustrativo

In [ ]:
```
from ACLIB.utils import modular_inverse, modexp, getprime
```

In [ ]:
```
def rsakeys(p,q):
    '''Returns pub and sec keys, e and d'''
    from ACLIB.utils import modular_inverse
    from random import randint
    phi = (p-1)*(q-1)
    e = 2*randint(1,phi>>1)+1
    while True:
        if e%p == 0 or e%q == 0:
            e = 2*randint(1,phi>>1)+1
            continue
        d = modular_inverse(e,phi)
        if d is None or d == e:
            e = 2*randint(1,phi>>1)+1
            continue
        return e,d
```

Numeri molto piccoli

In [ ]:
```
p = 19
q = 31
n = p*q
phi = (p-1)*(q-1)
e,d = rsakeys(p,q)
```

In [ ]:
```
e,d,n,phi
```

```
(23, 47, 589, 540)
```

In [ ]:
```
(e*d)%phi
```

```
1
```

In [ ]:
```
M = randint(1,n-1); M
```

```
412
```

In [ ]:
```
C = (M**e)%n; C
```

```
90
```

In [ ]:
```
(C**d)%n
```

```
412
```

Numeri piccoli per scopi crittografici

In [ ]:
```
from Crypto.Util import number
```

In [ ]:
```
N = 256    # Lunghezza in bit del modulo
```

In [ ]:
```
p = number.getPrime(N//2); p
```

```
222830599487371856893424612355349850299
```

In [ ]:
```
q = number.getPrime(N//2); q
```

```
187826365558767392152053768148528575479
```

In [ ]:
```
n = p*q
phi = (p-1)*(q-1)
e,d = rsakeys(p,q)
print(e,d,n)
```

```
39006778740733519911284238769189477943810633268029490928628277709978425518129 153763049986590027597276501721866417729070289943953163378366
73176771817886729 41853461636994392250829679061626014052158062520668353899216772841586372218221
```

In [ ]:
```
M = randint(1,n-1); M
```

```
92474819788637590179397021206150145576346415296034570800768126334482648597 46
```

In [ ]:
```
C = modexp(M,e,n); C
```

```
32273572003840310856543046487022381158721195186742039461809810717647849708442
```

In [ ]:
```
D = modexp(C,d,n)
D == M
```

```
True
```

Key management in Python con Crypto

In [ ]:
```
from Crypto.PublicKey import RSA
```

Generazione delle chiavi

In [ ]:
```
key = RSA.generate(2048)     # Generazione della chiave
```

In [ ]:
```
# Parametri fondamentali
p = key.p
q = key.q
n = key.n
e = key.e
d = key.d
```

In [ ]:
```
print(f"Public exponent e={e}")
```

In [ ]:
```
e==2**16+1  # 2**16+1 = 10000000000000001
```

La chiave pubblica, naturalmente, include solo i parametri pubblici

In [ ]:
```
public_key = key.publickey()
```

In [ ]:
```
public_key.n
```

In [ ]:
```
public_key.e
```

In [ ]:
```
public_key.d
```

Esportazione e importazione delle chiavi

- Esportazione della chiave pubblica

In [ ]:
```
# Per esportare la chiave pubblica: key.publickey().exportKey()
with open('mypubkey.pem','wb') as f:
    f.write(public_key.exportKey())
```

- Importazione della chiave pubblica

In [ ]:
```
imported_pub_key = RSA.importKey(open("mypubkey.pem").read())
```

In [ ]:
```
imported_pub_key == public_key
```

In [ ]:
```
public_key
```

Esportazione ed importazione della chiave privata

In [ ]:
```
# Per esportare la chiave privata è sufficiente: key.exportKey()
# Questo perché la chiave è la chiave privata! La chiave pubblica
# invece è un "sottoinsieme" che deve prima essere "estratto"
# usando il metodo publickey()
# E' però altamente consigliabile proteggere le chiave segreta con
# una strong passphrase
privateKey = \
key.exportKey(passphrase='_AIV3ryII5tr0ngIIIPa55w0rd_')
with open('myprivkey.pem','wb') as f:
    f.write(privateKey)
```

In [ ]:
```
with open("myprivkey.pem", "wb") as f:
    f.write(privateKey)
```

In [ ]:
```
imported_priv_key = \
   RSA.importKey(open("myprivkey.pem").read(),passphrase='_AIV3ryII5tr0ngIIIPa55w0rd_')
```

In [ ]:
```
imported_priv_key==key
```

Correttezza del protocollo RSA

- Ritorniamo ora alla questione lasciata in sospeso, e cioè la correttezza del protocollo
- Dobbiamo dimostrare che effettivamente Alice, con il calcolo di $C^d \bmod n$, rimette in chiaro il messaggio cifrato da Bob, ovvero che vale $C^d \bmod n = M$
- In uno dei passaggi algebrici, useremo la seguente uguaglianza:

$$ed = k \cdot \phi(n) + 1$$

per un qualche intero k.

- Ricordiamo al riguardo che d è l'inverso moltiplicativo di e (modulo $\phi(n)$), cioè che vale $e \cdot d \bmod \phi(n) = 1$.
- Questo vuol proprio dire che il prodotto $ed - 1$ è un multiplo di $\phi(n)$.

- Abbiamo dunque

$$\begin{aligned} C^d \bmod n &= (M^e \bmod n)^d \bmod n \\ &= M^{e \cdot d} \bmod n \\ &= M^{k \cdot \phi(n)+1} \bmod n \\ &= M \cdot M^{k \cdot \phi(n)} \bmod n \\ &= M \cdot M^{k \cdot (p-1) \cdot (q-1)} \bmod n \end{aligned}$$

- A questo punto dimostriamo preliminarmente che vale l'uguaglianza:

$$M \cdot M^{k \cdot (p-1) \cdot (q-1)} \bmod p = M \bmod p$$

- L'uguaglianza è immediatamente verificata se $M \bmod p = 0$, perché in tal caso entrambi i membri sono ovviamente 0
- In caso contrario, cioè se $M \bmod p \neq 0$, abbiamo

$$\begin{aligned} \left(M \cdot M^{k \cdot (p-1) \cdot (q-1)}\right) \bmod p &= \left(M \cdot \left(M^{p-1}\right)^{k \cdot (q-1)}\right) \bmod p \\ &= \left(M \cdot \left(M^{p-1} \bmod p\right)^{k \cdot (q-1)}\right) \bmod p \\ &= \left(M \cdot (1)^{k \cdot (q-1)}\right) \bmod p \\ &= M \bmod p \end{aligned}$$

- Allo stesso modo vale evidentemente

$$M \cdot M^{k \cdot (p-1) \cdot (q-1)} \bmod q = M \bmod q$$

- Dunque vale simultaneamente $C^d \bmod p = M \bmod p$ e $C^d \bmod q = M \bmod q$.
- In altri termini le quantità C^d e M hanno gli stessi resti nella divisione per p e per q.
- Ma allora, necessariamente poiché p e q sono relativamente primi e $n = pq$ (si ricordi il Teorema cinese dei resti), vale anche $C^d \bmod n = M \bmod n$

Efficienza

- La relativa "abbondanza" dei numeri primi e la mancanza di ulteriori requisiti forti (tipo che il numero generato sia un safe prime) fa sì che p e q possano essere determinati velocemente.
- Il calcolo di e e poi del suo inverso sono eseguiti una volta sola
- Cifratura e decifrazione richiedono il calcolo di esponenziali modulari. In se si tratta di operazioni efficientemente implementate ma che coinvolgono numeri non gestibili in aritmetica di macchina.
- Sulla scelta di e ritorneremo più avanti in relazione alla sicurezza. Qui ci interessa capire come si può procedere per trovare un intero coprimo con $\phi(n)$.
- Una semplice scelta consiste nel prendere e primo e maggiore di $\max\{p, q\}$.
- Infatti $\phi(n) = (p-1) \cdot (q-1)$ e dunque i suoi fattori primi sono inferiori a p e q.
- Altrimenti si può procedere per tentativi "casuali", grazie al fatto che il calcolo del MCD è molto veloce.

- Un'ulteriore possibilità consiste nel fissare a priori un esponente piccolo (es 3 o 5) e continuare a generare i primi p e q fino a quando non si verifica la condizione $\mathrm{MCD}(e, (p-1) \cdot (q-1)) = 1$.
- Questo approccio ha alcuni vantaggi in ordine alla velocizzazione del calcolo di $M^e \mod n$, cosa che si potrebbe rivelare utile in applicazioni della firma digitale (come vedremo).
- Nelle implementazioni "reali" del protocollo si procede in questo modo ma il valore di e che viene scelto è più elevato.

Sicurezza

- Dal punto di vista "strettamente matematico", la sicurezza di RSA risiede nella difficoltà di fattorizzare numeri interi di grandi dimensioni.
- Se la fattorizzazione fosse facile (cioè se disponessimo di un algoritmo di fattorizzazione efficiente) allora RSA sarebbe violabile perché, recuperando p e q, Eva potrebbe calcolare $\phi(n)$ e quindi anche d.
- Il viceversa non è dimostrato ma, in più di quarant'anni di sforzi, non è stato trovato un metodo efficiente per violare RSA, che quindi non passi attraverso la fattorizzazione del modulo n.
- Altro faccenda è però come vada implementato RSA in modo che non presti il fianco ad attacchi "vincenti".

Alcuni possibili problemi (generali e/o specifici del protocollo base)

Malleabilità

- Ricordiamo che un algoritmo di cifratura $\mathbf{E}$ è malleabile se, dato un testo cifrato $C_1 = \mathbf{E}(M_1)$, è possibile creare un secondo testo cifrato C_2 tale che $C_2 = \mathbf{E}(M_2)$ ed M_2 sia in relazione "significativa" con M_1 (per semplicità, nella notazione abbiamo omesso di indicare la chiave).
- Il protocollo RSA di base è malleabile.
- Infatti, il prodotto di due messaggi cifrati corrisponde al prodotto dei due messaggi in chiaro.
- Questo fatto presta il fianco al seguente attacco nel modello Chosen ciphertext.

- Sia $C_1 = M_1^e \bmod n$ il testo cifrato.
- Creiamo un testo cifrato "intermedio" $C_I = 2^e \bmod n$.
- Questo naturalmente possiamo farlo perché la chiave e è pubblica.
- Ora creiamo C_2 nel modo seguente:

$$C_2 = C_1 \cdot C_I$$

- Questo è il nostro chosen ciphertext.
- Se riusciamo, con una qualsiasi ragione, a far decifrare C_2 abbiamo facilmente modo di ottenere M_1. Infatti:

$$\begin{aligned} C_2^d \bmod n &= (C_1 \cdot C_I)^d \bmod n \\ &= ((M_1^e \bmod n) \cdot (2^e \bmod n))^d \bmod n \\ &= \left((M_1^e)^d \cdot (2^e)^d\right) \bmod n \\ &= (M_1 \cdot 2) \bmod n \end{aligned}$$

- Basta quindi un semplice shift del messaggio decifrato per ottenere quello originale

Caso di e troppo piccolo

- Un secondo problema occorre quando l'esponente e è piccolo e i messaggi sono a loro volta brevi
- Ad esempio, se $e = 3$ e il messaggio è breve, può acccadere che $M^3.
- In questo caso la riduzione modulo n è inutile e dunque $C = M^3$.
- In questo caso Eva può rimettere in chiaro il messaggio semplicemente calcolando la radice cubica di C.
- Per questa ragione, lo standard FIPS prevede che e sia non minore di $2^{16} + 1 = 65537$.

Possibilità di fattorizzare n

- Esiste poi tutta una "gamma" di problemi legati alla possibilità di fattorizzare n se i numeri primi p e q hanno determinate caratteristiche "sbagliate"
- Presentiamo qui un caso che non richiede particolare sofisticazione matematica
- Il caso riguarda la situazione in cui p e q sono "molto vicini" (affermazione che ovviamente dovremo andare a quantificare)
- Ricordiamo solo due semplici fatti, sicuramente già noti.

1. Se $n = p \cdot q$, e se $p \neq q$, allora necessariamente (assumendo senza perdita di generalità che $p > q$) vale

$$q < \sqrt{n} < p$$

2. Il prodotto di due numeri dispari, come è il caso di n, può sempre essere espresso come differenza di quadrati di numeri interi. Posto infatti $n = p \cdot q$ (con p e q interi dispari) risulta

$$n = \left(\frac{p+q}{2}\right)^2 - \left(\frac{p-q}{2}\right)^2 = \frac{1}{4}\left((p+q)^2 - (p-q)^2\right)$$

3. Ad esempio: $11 \cdot 7 = 77 = 9^2 - 2^2$

- Se dunque riuscissimo ad esprimere il modulo RSA n come differenza di quadrati

$$n = x^2 - y^2 = (x + y)(x - y)$$

necessariamente (cioè poiché p e q sono primi) avremmo $p = x + y$ e $q = x - y$

- Nel caso appena visto $77 = (9 + 2)(9 - 2) = 11 \cdot 7$
- Nel caso almeno uno dei due fattori sia a sua volta un numero composto si possono avere anche decomposizioni diverse (naturalmente)
- Ad esempio, nel caso di $n = 945$ abbiamo

$$n = 27 \cdot 35 = 31^2 - 4^2$$

ma anche

$$n = 105 \cdot 9 = 57^2 - 48^2$$

- Volendo portare un attacco a RSA tramite la fattorizzazione di n dobbiamo chiederci con quanta facilità si possono trovare le due quantità $\frac{p+q}{2}$ e $\frac{p-q}{2}$, ovviamente senza la conoscenza di p e q.
- L'idea parte dalla riscrittura di $n = x^2 - y^2$ nel modo seguente

$$x^2 - n = y^2$$

che suggerisce la strategia brute force illustrata di seguito

- Si inizia con un certo valore $x = x_0$ e si calcola $z_0 = x_0^2 - n$.
- Se z_0 è un quadrato perfetto, stop, abbiamo finito, altrimenti procediamo incrementando il valore x_0
- In generale, al generico passo i, posto $x_i = x_0 + i$, se $z_i = x_i^2 - n$ è un quadrato perfetto, stop, altrimenti procediamo con un ulteriore incremento
- Dobbiamo ovviamente chiarire alcuni punti. (1) Come scegliere il valore x_0 e (2) qual è la complessità di questo procedimento.

- Riguardo il punto 1, dall'uguaglianza

$$n = \left(\frac{p+q}{2}\right)^2 - \left(\frac{p-q}{2}\right)^2$$

ricaviamo ovviamente

$$\left(\frac{p+q}{2}\right)^2 - n = \left(\frac{p-q}{2}\right)^2$$

- È chiaro quindi che il valore iniziale x_0 deve essere non maggiore di $\frac{p+q}{2}$ perché altrimenti avremmo $z_0 = x_0^2 - n > \left(\frac{p+q}{2}\right)^2 - n = \left(\frac{p-q}{2}\right)^2$ e poiché il valore di x_i (e dunque di z_i) aumenta ad ogni passo, l'uguaglianza non verrebbe mai soddisfatta
- È poi facile verificare che $p \neq q$ e $p \cdot q = n$ implica $\frac{p+q}{2} > \sqrt{n}$ (basta elevare al quadrato ed eseguire pochi passaggi algebrici).
- Dunque il punto di partenza dell'algoritmo può essere proprio $x_0 = \sqrt{n}$, o meglio $x_0 = \lceil \sqrt{n} \rceil$.

- Possiamo quindi scrivere l'algoritmo completo
- Dato n, prodotto di due numeri dispari distinti, poni $x = \lceil \sqrt{n} \rceil$
- Calcola $z = x^2 - n$
- Se z non è un quadrato perfetto, poni $x = x + 1$ e ritorna al passo 2
- Calcola e restituisci $x + \sqrt{z}$ e $x - \sqrt{z}$

In [ ]:
```python
from ACLIB.utils import intsqrt
def RSAfactorize(n,printiter=False):
    '''n deve essere il prodotto di due interi dispari. Usa il metodo di Fermat'''
    x = intsqrt(n)+1   # Guess iniziale
    z = x*x-n
    s = intsqrt(z)
    i = 0
    while s*s != z:   # Se z è un quadrato perfetto, stop
        i += 1
        z += (x<<1)+1 # Altrimenti calcoliamo z = (x+1)(x+1)-n (= z attuale+2x+1)
        x += 1
        s = intsqrt(z)
    if printiter:
        print(f"Numero di iterazioni: {i}")
    return x+s,x-s
```

In [ ]:
```python
from Crypto.Util import number
```

In [ ]:
```python
n = number.getPrime(32)*number.getPrime(32)
print(f"Numero da fattorizzare: {n}")
p,q=RSAfactorize(n)
print(p,q,p*q==n)
```

```
Numero da fattorizzare: 8427322075335414049
2973192269 2834435621 True
```

Efficienza del metodo

- Rimane ancora da discutere la questione di quando sia fattibile applicare l'algoritmo, ovvero sotto quali condizioni esso riesce a fattorizzare l'input in "tempi ragionevoli".
- Abbiamo già anticipato che questo motodo ha chance di successo (su numeri di grandi dimensioni) se p e q sono vicini.

  È il momento di quantificare l'affermazione.

- Supponiamo che valga la seguente diseguaglianza:
$$\frac{p-q}{2} < c \cdot \sqrt{2 \cdot q}$$
con c costante.
- Dall'uguaglianza da cui siamo partiti per lo sviluppo dell'algoritmo, e cioè
$$\left(\frac{p+q}{2}\right)^2 - n = \left(\frac{p-q}{2}\right)^2$$
possiamo ricavare
$$\left(\frac{p+q}{2} + \sqrt{n}\right) \cdot \left(\frac{p+q}{2} - \sqrt{n}\right) = \left(\frac{p-q}{2}\right)^2$$
- Poiché $q < \sqrt{n} < p$, possiamo minorare con $2q$ il primo fattore a sinistra (sostituendo cioè sia p che $\sqrt{n}$ con q
- Per ipotesi, poi, possiamo maggiorare il membro destro con $2qc^2$ e dunque, in definitiva, abbiamo
$$2 \cdot q \cdot \left(\frac{p+q}{2} - \sqrt{n}\right) < 2 \cdot q \cdot c^2$$

ovvero

$$\frac{p+q}{2} - \sqrt{n} < \frac{2 \cdot q \cdot c^2}{2 \cdot q} = c^2$$

- Questo è il risultato desiderato perché la quantita a sinistra è proprio il numero passi (iterazioni) chel'algoritmo impiega per arrivare alla soluzione e la diseguaglianza dice che questo è limitato da c^2.
- In concreto, però, che cosa significa la assunzione che abbiamo fatto per ottenere questo risultato? Ovvero che:

$$\frac{p-q}{2} < c \cdot \sqrt{2 \cdot q}$$

- Poiché q è $O(\sqrt{n})$, la nostra assunzione implica che la differenza fra p e q risulta $O(\sqrt[4]{n})$.
- In termini di cifre, questo a sua volta significa che p e q coincidono nella metà più significativa.

Vediamo qualche esempio

In [ ]:
```
from ACLIB.utils import intsqrt
from math import sqrt
p = number.getPrime(1024)
q = (p - 10*intsqrt(p))|1  # q è tentativamente p-10 sqrt(p)
while not number.isPrime(q):
    q -= 2
print(f"Fattore p (maggiore): {p}")
print(f"Fattore q (minore):   {q}")
print(f"Differenza:           {p-q}")
# Numero di passi previsto
f = 1/(2*intsqrt(2*q))
c = int((p-q)*f)
print(f"Numero approssimativo di passi:", c*c)
```

```
Fattore p (maggiore): 1140999275548968718795489807202012617079376747031085343976724803877998444315503292610413573417184232000572320127581393972296460292838412662062127511512281909795482864596752450979289943469585361026793651497303823398686837737506406931585325717903826386728175659934949905252163878080569721802636269504535565510281
Fattore q (minore):   1140999275548968718795489807202012617079376747031085343976724803877998444315503292610413573417184232000572320127581393972296460292838412662062127511512280841619806513104128370695167082030879911268804283062327734879018554389891045410374791230696160060013137757184207180064321420286154835744575939077897073296037
Differenza:           1068175676351492624080284122861438705449757989368434976088519668283347615361521210534487207666326715037902750742725187317360519542382281786755967458581806780
Numero approssimativo di passi: 9
```

In [ ]:
```
n = p*q
r,s = RSAfactorize(n,printiter=True)
print(r,s,r*s == n)
```

```
Numero di iterazioni: 12
1140999275548968718795489807202012617079376747031085343976724803877998444315503292610413573417184232000572320127581393972296460292838412662062127511512281909795482864596752450979289943469585361026793651497303823398686837737506406931585325717903826386728175659934949905251638780805697218026362695045355655102812140999275548968718795489807202012617079376747031085343976724803877998444315503292610413573417184232000572320127581393972296460292838412662062127511512280841619806513104128370695167082030879911268804283062327734879018554389891045410374791230696160060013137757184207180064321420286154835744575939077897073296037 True
```

In [ ]:
```
p==r
```

```
True
```

Vulnerabilità dipendenti dai generatori casuali

- In uno studio condotto nel 2012, sono stati analizzati 4.7 milioni di moduli RSA di 1024 bit.
- È risultato che i duplicati non sono infrequenti.
- In 12720 casi, i moduli avevano un fattore primo in comune.
- Per quanto riguarda la duplicazione del modulo, vedremo come sia possibile utilizzare questa circostanza per ricostruire le corrispondenti chiavi private.
- Riguardo l'altra situazione, il fatto cioè che due moduli abbiano un fattore primo in comune, è immediato vedere come questo renda vulnerabile il protocollo.
- Infatti, il fattore in comune può essere scoperto calcolando il MCD dei due moduli.
- Da questo e dal modulo si può poi banalmente risalire alla determinazione dell'altro fattore.

$$n_1 = p \cdot q_1, \quad \text{e} \quad n_2 = p \cdot q_2 \qquad \Rightarrow \qquad p = \mathrm{MCD}(n_1, n_2)$$

- La causa di queste situazioni estremamente pericolose è da ascrivere primariamente a debolezze dei generatori casuali coinvolti nella scelta dei numeri primi.
- Riguardo la condivisione del modulo nella sua interezza esiste in letteratura tutta una serie di possibili attacchi che sfruttano altre debolezze, vuoi di natura tecnica (es. esponente privato relativamente piccolo), vuoi legate ad un possibile scenario applicativo "azzardato".
- Nel seguito riportiamo (per studio opzionale) un caso di quest'ultimo tipo

Un attacco al modulo comune

- Consideriamo uno scenario applicativo in cui è presente un amministratore o, più in generale, una trusted central authority, e un insieme di k utenti.
- Supponiamo che tale authorithy, riconosciuta e affidabile, distribuisca agli utenti altrettante coppie di chiavi, $(e_i, d_i), i = 1, \ldots, k$.
- Le chiavi, condividono lo stesso modulo n, anch'esso distribuito agli utenti ma ovviamente senza i fattori p e q.
- Questo può sembrare conveniente perché la condivisione del modulo può comportare vantaggi di natura computazionale e, a prima vista, nessun problema di sicurezza.
- Tuttavia questo è un errore. Infatti, dalla conoscenza di una coppia e_j e d_j si può risalire alla fattorizzazione di n e quindi alla determinazione di tutti gli esponenti privati.

- Con $n = p \cdot q$, consideriamo il gruppo $\mathbf{Z}_n^*$.
- Gli elementi, come sappiano, sono gli interi minori di n che sono primi con n (perché solo questi hanno inverso).
- Sappiamo anche che il numero di elementi è $\phi(n) = (p-1) \cdot (q-1)$.
- Come esempio, proviamo a scrivere la tabella del prodotto per gli elementi del gruppo nel caso $n = p \cdot q = 3 \cdot 5 = 15$.

	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

- Possiamo notare che nella diagonale della tabella ci sono quattro 1.
- Questo vuol dire che ci sono quattro elementi del gruppo che soddisfano l'equazione $x^2 \bmod 15 = 1$, ovvero che sono radici quadrate dell'unità.
- Questa è una proprietà generale dei gruppi $\mathbf{Z}_n^*$ in cui n è il prodotto di due primi.
- In questo caso, conoscendo i due primi p e q, è semplice trovare le quattro radici dell'unità, usando la seguente formula, che altro non è il Teorema cinese dei resti specializzato al caso di due primi:

$$x = \left(\left(q\left(q^{-1} \bmod p\right)\right) x_p + \left(p\left(p^{-1} \bmod q\right)\right) x_q\right) \bmod n$$

dove $x_p, x_q \in \{(1,1), (1, q-1), (p-1, 1), (p-1, q-1)\}$ sono tutte le possibili coppie in cui gli elementi sono, rispettivamante, radici quadrate dell'unità modulo p e modulo q

---

- Verifichiamo la formula proprio con $p = 3$ e $q = 5$
- In questo caso abbiamo

$$q\left(q^{-1} \bmod p\right) = 5\left(5^{-1} \bmod 3\right) = 5 \cdot 2 = 10$$

e

$$p\left(p^{-1} \bmod q\right) = 3\left(3^{-1} \bmod 5\right) = 3 \cdot 2 = 6$$

da cui, usando tutte le possibilità per x_p e x_q otteniamoeffettivamente le quattro radici quadrati dell'unità in $\mathbf{Z}_{15}^*$:

1. $x_1 = (10 \cdot 1 + 6 \cdot 1) \bmod 15 = 16 \bmod 15 = 1$
2. $x_2 = (10 \cdot 1 + 6 \cdot 4) \bmod 15 = 34 \bmod 15 = 4$
3. $x_3 = (10 \cdot 2 + 6 \cdot 1) \bmod 15 = 26 \bmod 15 = 11$
4. $x_4 = (10 \cdot 2 + 6 \cdot 4) \bmod 15 = 44 \bmod 15 = 14$

- Chiamiamo banali le due radici x_1 e x_4
- Le due radici non banali, x_2 e x_3, corrispondono alle coppie $(1, q-1)$ e $(p-1, 1)$, ed è immediato verificare che:

1. per x_2 vale $(x_2 - 1) \bmod p = 0$ e $(x_2 - 1) \bmod q \neq 0$
2. per x_3 vale l'esatto contrario, cioè $(x_3 - 1) \bmod p \neq 0$ e $(x_3 - 1) \bmod q = 0$

- Questo vuol dire che $x_2 - 1$ è multiplo di p ma non di q e dunque neppure di n.
- Ma allora $\mathrm{MCD}(x_2 - 1, n) = p$
- Simmetricamente, $\mathrm{MCD}(x_3 - 1, n) = q$

- Se dunque conoscessimo un modo efficiente per calcolare una radice non banale dell'unità, potremmo calcolare un fattore di n e, quindi, chiaramente, fattorizzare completamente il modulo
- Le radici banali invece non aiutano (e dovrebbe essere immediato il perché).
- In effetti, disponiamo di un algoritmo randomizzato che, usando e e d, riesce a trovare una radice dell'unità di $\mathbf{Z}_n^*$.
- L'algoritmo non ha controllo su quale sia la radice viene trovata, che potrebbe quindi anche essere una delle due banali
- Tuttavia, poiché le radici non banali sono la metà, ripetendo l'algoritmo è ragionevole ritenere che con pochi tentativi si riesca a individuare una delle radici "buone".

Fattorizzazione di n tramite e e d

- Nell'algoritmo facciamo uso della seguente proprietà, che abbiamo già usato nella dimostrazione di correttezza dell'RSA, e che qui riscriviamo per comodità in modo leggermente diverso

> Poiché $e \cdot d = k \cdot \phi(n) + 1 \bmod n$, se z è un numero relativamente primo con n risulta
> $z^{e \cdot d - 1} \bmod n = z^{k \cdot (p-1) \cdot (q-1)} \bmod n = 1$

1. Fattorizziamo parzialmente la quantità $e \cdot d - 1 = k \cdot \phi(n) = k \cdot (p-1) \cdot (q-1)$ nel modo seguente:
$$e \cdot d - 1 = 2^t \cdot r$$
in cui $t \geq 2$ perché $p-1$ e $q-1$ sono numeri pari e r è un numero dispari. La fattorizziamo può essere ottenuta facilmente dividendo per due finché non c'è resto (t è proprio il numero di volte in cui abbiamo eseguito la divisione per 2).
2. Scegliamo un numero z a caso nell'insieme $\{1, 2, \ldots, n-1\}$ e calcoliamo $\mathrm{MCD}(z, n)$. Se siamo così fortunati che tale valore è maggiore di 1, abbiamo evidentemente già trovato un divisore di n (o p o q) e dunque abbiamo finito. Altrimenti z è primo con n e per esso dunque vale $z^{e \cdot d - 1} \bmod n = 1$.
3. Calcoliamo ora $x = z^r \bmod n$. Questa quantità non può essere 1 perché r è dispari. Tuttavia, poiché (come abbiamo appena visto) $x^{2^t} = (z^r)^{2^t} = z^{r 2^t} = z^{e \cdot d - 1} = 1 \bmod n$, abbiamo la certezza che, partendo da x ed elevando ripetutamente al quadrato, prima o poi otteniamo il valore 1
4. Posto $x_0 = x$, calcoliamo cioè la sequenza di valori: $x_{i+1} = x_i^2 \bmod n$, $i = 0, 1, \ldots$ e ci fermiamo non appena troviamo $x_{i+1} = 1$. In questo caso $\bar{x} = x_i$ è una radice quadrata dell'unità in $\mathbf{Z}_n^*$.
5. Effettuiamo la verifica $\mathrm{MCD}(\bar{x} - 1, n)$. Se questa quantità è maggiore di 1 e diversa da n, allora $\bar{x}$ è una delle due radici non banali e abbiamo trovato uno dei due fattori di n. Altrimenti $\bar{x}$ è una delle due radici banali e ripetiamo il procedimento scegliendo un altro valore di z.

Implementazione in Python

In [ ]:
```
from Crypto.Random.random import randint
def RSAattack(e,d,n):
    '''Restituisce i due fattori primi, p e q, tali che n=pq.
       Mostra come la condivisione del modulo fra piu coppie
       di esponenti (e_i,d_i) consenta, proprio attraverso
       la fattorizzazione di n, di ricostruire tutte le
       chiavi private che condividono il modulo.
    '''
    r = e*d-1
    t = 0
    while r&1==0:
        r = r>>1
        t+=1
    while True:
        z = randint(1,n-1)
        f = Euclid(z,n)
        if f>1 and f!=n:
            return f,n//f
        x = modexp(z,r,n)
        x2 = modexp(x,2,n)
        while x2 != 1:
            x = x2
            x2 = modexp(x,2,n)
        f = Euclid(x-1,n)
        if f>1 and f!=n:
            return f,n//f
```

- Un esempio con chiavi reali

In [ ]:
```
from ACLIB.utils import Euclid,modexp
```

In [ ]:
```
key = RSA.generate(2048)
p = key.p
q = key.q
n = key.n
e = key.e
d = key.d
```

In [ ]:
```
p1,q1 = RSAattack(e,d,n)
```

In [ ]:
```
(p1==p and q1==q) or (p1==q and q1==p)
```

Aspetti implementativi di RSA

- Alla luce di quanto abbiamo visto, possiamo affermare che la sicurezza delle cifrature con l'algoritmo RSA richiede che siano considerati molti aspetti apparentemente marginali per non incorrere in vulnerabilità dall'effetto più o meno disastroso.
- In particolare, abbiamo visto quanto segue (che però non è tutto...)
- I generatori casuali utilizzati per generare i fattori primi del modulo devono certamente essere ottimi dal punto di vista teorico. Questo però non basta; la generazione deve essere effettuata dopo che sia stata raccolta sufficiente entropia da garantire che le eventuali collisioni non abbiano probabilità maggiore di quanto previsto teoricamente.
- I fattori primi p e q non devono essere troppo vicini, per evitare l'attacco basato sull'algoritmo di fattorizzazione di Fermat.
- Moduli condivisi devono essere sistematicamente evitati.
- Gli esponenti e e d non devono essere piccoli da rendere inefficace la riduzione modulo n.
- La malleabilità del protocollo deve essere eliminata con opportuni accorgimenti.

- Ci occuperemo ora proprio di quest'ultimo aspetto, che non abbiamo ancora preso in considerazione.
- Prima però, discutiamo un paio di questioni legati all'efficienza computazionale, perché anch'essa fa ovviamente parte degli obiettivi di una implementazione reale.

Qualche "trucco" per migliorare l'efficienza di RSA

- Il primo aspetto riguarda la scelta dell'esponente pubblico e.
- La raccomandazione FIPS, come già sottolineato, è che esso sia non minore di 65537.
- In molte implementazioni concrete la scelta cade proprio su questo numero perché si tratta di un intero primo la cui rappresentazione binaria comprende quasi esclusivamente cifre 0.
- Infatti $65537 = 2^{16} + 1$ e la rappresentazione binaria è 10000000000000001.
- Proprio la presenza di molte cifre 0 rende più efficiente il calcolo delle esponenziali modulari.

- Il secondo aspetto implementativo riguarda la fase di decifrazione.
- Se C è il messaggio cifrato, l'algoritmo (come sappiamo) prevede di calcolare $M = C^d \bmod n$.
- La decifrazione può essere svolta in modo più efficiente usando il teorema cinese dei resti e riducendo poi la dimensione degli esponenti

1. Per il teorema cinese dei resti sappiamo che, disponendo delle due quantità
$$M_p = C^d \bmod p \qquad \text{e} \qquad M_q = C^d \bmod q,$$
possiamo risalire al valore M nel modo seguente:
$$M = \left(\left(q \cdot (q^{-1} \bmod p)\right) M_p + \left(p \cdot (p^{-1} \bmod q)\right) M_q\right) \bmod n$$
in cui i coefficienti $q \cdot (q^{-1} \bmod p)$ e $p \cdot (p^{-1} \bmod q)$ possono essere precalcolati
2. Tuttavia, anziché calcolare M_p e M_q nel modo ovvio, determiniamo preliminarmente le quantità
$s = d \bmod (p-1)$ e $t = d \bmod (q-1)$, anch'esse indipendenti da qualsiasi messaggio
   - Si noti che, per definizione, $d = s + a \cdot (p-1)$ (per un qualche intero a) e dunque
   $C^d \bmod p = C^{s+a\cdot(p-1)} \bmod p = C^s \bmod p$.
   - Analogamente, $y^d \bmod q = y^t \bmod p$

   e quindi calcoliamo M_p e M_q nel modo seguente
$$M_p = C^s \bmod p \qquad \text{e} \qquad M_q = C^t \bmod q$$

- In questo modo, a parte le quantità che vengono calcolate preliminarmente (una sola volta), il calcolo richiede due esponenziazioni anziché una sola, ma su numeri di lunghezza circa dimezzata.
- Oltre a ciò vengono calcolati due prodotti e una somma, operazioni di costo asintotico inferiore

In [ ]:
```
from ACLIB.utils import modprod, modular_inverse, modexp
```

In [ ]:
```
key = RSA.generate(2048)
s = key.d%(key.p-1)
t = key.d%(key.q-1)
cp = key.q*modular_inverse(key.q,key.p)
cq = key.p*modular_inverse(key.p,key.q)
```

In [ ]:
```
def RSAdecrypt(y,n,p,q,s,t,cp,cq):
    xp = modexp(y,s,p)
    xq = modexp(y,t,q)
    x = (modprod(cp,xp,n)+modprod(cq,xq,n))%n
    return x
```

In [ ]:
```
M=2
y = modexp(M,key.e,key.n); y
```

```
96149045257633610371526440739647181735619648620277846097169772523744191315241839097090839449324818424416485294990934377786633255434344716006610448677188677423603032052337054576913839800221634116920052773967577444908744170783303462143005060382179291415793246138539851479563892178916703604636814013321840930779529414989244547899073186674438422208796338312287827909649431581844044878453311624098126208825464987033117919729534146709989928113520486689734082484089955883756537706569180557530042488649765439810313458865436390521944653441010155132120560219450233041524481177879210057244413720064948569669572037168728501909 74
```

In [ ]:
```
modexp(y,key.d,key.n)
```

```
2
```

In [ ]:
```
RSAdecrypt(y,key.n,key.p,key.q,s,t,cp,cq)
```

```
2
```

In [ ]:
```
from time import time
```

In [ ]:
```
n = 10
startt = time()
for _ in range(n):
    RSAdecrypt(y,key.n,key.p,key.q,s,t,cp,cq)
endt = time()
print(f"Elapsed time: {endt-startt:.4f}")
```

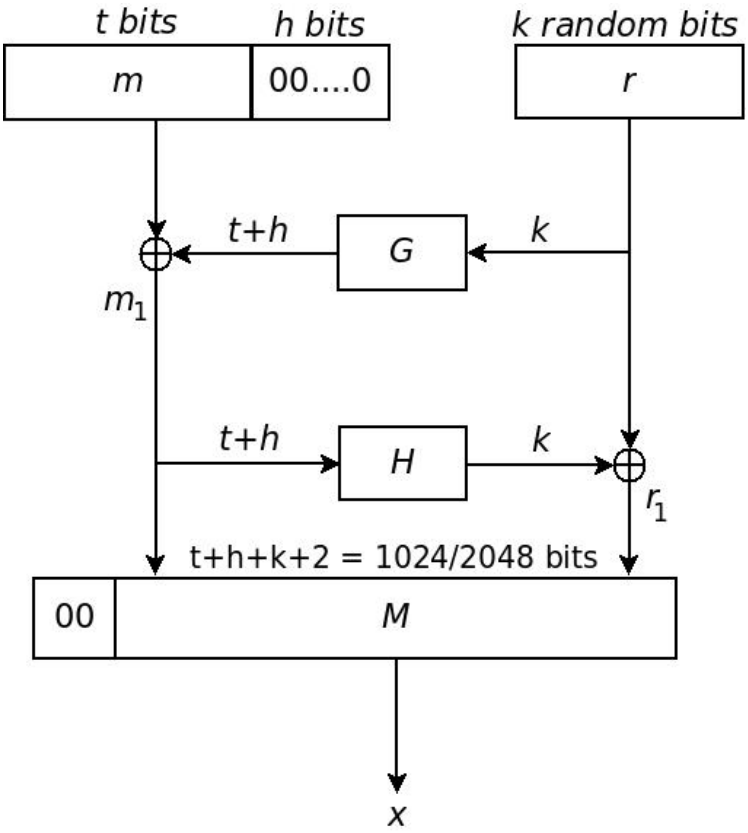
```
Elapsed time: 7.2141
```

In [ ]:
```
startt = time()
for _ in range(n):
    modexp(y,key.d,key.n)
endt = time()
print(f"Elapsed time: {endt-startt:.4f}")
```

```
Elapsed time: 28.6519
```

Optimal Asymmetric Encryption Padding (OAEP)

- Il modo in cui la cifratura RSA viene effettuata in concreto (non solo nell'RSA, ma praticamente in tutti gli algoritmi di cifratura, simmetrici o asimmetrici) include informazione addizionale chiamata padding.
- Per RSA il padding è costituito da sequenze (pseudo)casuali e dunque, anche per questo scopo, è necessario disporre di generatori affidabili.
- Oltre al generatore, il funzionamento dello schema di padding OAEP necessita la disponibilità di due funzioni hash crittografiche, che indicheremo genericamente con G e H.
- Se N indica la lunghezza in bit del modulo, la lunghezza massima "utile" dei messaggi è $t = N - h - k - 2$, dove h e k sono parametri del protocollo.
- Per gli scopi che vedono l'utilizzo di RSA, tale "spazio" è in generale più che sufficiente, dato che la lunghezza tipica moduli, espressa in byte, è non inferiore a 256
- Il seguente diagramma illustra il funzionamento di OAEP.
- Il messaggio finale, visto come sequenza di bit (o di byte), viene poi convertito nel numero x che costituisce l'input per l'algoritmo RSA.



- La decifrazione segue il percorso inverso.
- Dapprima (con l'algoritmo RSA) x viene ricalcolato e poi trasformato nella sequenza di byte M.
- La funzione H può quindi essere applicata allo stesso input m_1 della fase di cifratura.
- La seconda operazione $\oplus$ in fase di cifratura aveva fornito $r_1 = H(m_1) \oplus r$; grazie alle proprietà di $\oplus$ possiamo ora ottenere $r = r_1 \oplus H(m_1)$.
- Allo stesso modo, da $m_1 = m \oplus G(r)$ possiamo ora ottenere $m = m_1 \oplus G(r)$.
- Se gli ultimi h bit di m non sono 0, il messaggio viene rigettato.

Un esempio "completo"

- In questo esempio, il receiver è Bob e dunque è lui che genera le chiavi e le esporta

In [ ]:
```
################### Generazione delle chiavi ####################
from Crypto.PublicKey import RSA
key = RSA.generate(2048)
# Esportazione della chiave pubblica su file
with open('BobKey.pem','wb') as f:
    f.write(key.publickey().exportKey())
# Esportazione della chiave privata su file con password di protezione
privateKey = \
key.exportKey(passphrase='_AIV3ryII5tr0ngIIIPa55w0rd_')
with open('BobSecretKey.pem','wb') as f:
    f.write(privateKey)
```

In [ ]:
```
################## Cifratura del messaggio ######################
from Crypto.Cipher import AES, PKCS1_OAEP
from Crypto.Random import get_random_bytes
message = 'Questo è un messaggio'.encode('utf-8')
# Importazione della chiave del ricevente
BobKey = RSA.importKey(open("BobKey.pem").read())
# Creiamo un oggetto per la preparazione e la cifratura secondo lo schema OAEP-RSA
rsa = PKCS1_OAEP.new(BobKey)
# Generiamo ora una chiave simmetrica
symmetricKey = get_random_bytes(16)
# Cifriamo la chiave simmetrica
rsaEncryptedSymmKey = rsa.encrypt(symmetricKey)
# Cifriamo il messaggio usando AES con la chiave simmetrica
IV = get_random_bytes(16)
aes = AES.new(symmetricKey, AES.MODE_CFB, IV)
encMessage = IV+aes.encrypt(message)
# Inviamo la coppia formata dalla chiave simmetrica cifrata con RSA
# e il messaggio cifrato con la medesima chiave simmetrica
toBob = (rsaEncryptedSymmKey,encMessage)
```

In [ ]:
```
################## Decifrazione del messaggio ######################
# from Crypto.PublicKey import RSA
rsaEncryptedSymmKey,encMessage = toBob
# Recuperiamo la chiave RSA privata
with open('BobSecretKey.pem','r') as f:
    key = f.read()
privateKey = RSA.importKey(key,passphrase='_AIV3ryII5tr0ngIIIPa55w0rd_')
# La prima cosa da fare è decifrare il messaggio cifrato con RSA per recuperare la
# chiave simmetrica
rsa = PKCS1_OAEP.new(privateKey)
symmetricKey = rsa.decrypt(rsaEncryptedSymmKey)
# Recuperiamo l'IV e decifriamo ora il messaggio usando AES
IV = encMessage[:16]
aes = AES.new(symmetricKey, AES.MODE_CFB, IV)
decryptedMessage = aes.decrypt(encMessage)[16:]
```

In [ ]:
```
decryptedMessage.decode('utf-8')
```