Algoritmi di Crittografia (2023/24)

Notebook 6

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

Latex definitions

- In questo notebook presentiamo i sistemi asimmetrici proposti da M.O. Rabin e T. El Gamal
- Si tratta di due sistemi che non hanno guadagnato molta importanza, perché "oscurati" da RSA.
- El Gamal è interessante perché mostra come solo un "piccolo passo avanti", rispetto al protocollo di scambio di chiavi proposto da Diffie ed Hellman, avrebbe prodotto il primo vero protocollo asimmetrico (qualche anno prima di RSA)
- Anche la sicurezza del protocollo di El Gamal è dunque basata sul logaritmo discreto
- Il protocollo di Rabin è interessante perché, a differenza di RSA, è "dimostrabilmente" equivalente alla fattorizzazione, su cui basa la propria sicurezza
- Vedremo però che ha un "difetto" che ne ha sbarrato la strada all'impiego pratico

Il sistema crittografico asimmetrico di Rabin

- Di poco posteriore alla pubblicazione di RSA, il sistema inventato da Michael O. Rabin ha la ragguardevole proprietà teorica di essere equivalente alla fattorizzazione
- Più precisamente, un algoritmo polynomial time in grado di violare il sistema di Rabin potrebbe essere utilizzato per la fattorizzazione di interi (e viceversa)
- Possiede però anche una limitazione, che ne ha impedito la diffusione
- La decifrazione produce infatti quattro possibili candidati come testo in chiaro corrispondente ad un dato ciphertext e, se il testo originale è un numero o una sequenza di bit, questo può essere un serio problema

Un esempio numerico preliminare ralativo al calcolo delle radici quadrate

In [ ]:
```
from random import randint
from ACLIB.utils import modular_inverse, Euclid, getprime
```

- Vogliamo calcolare le radici quadrate di un numero y in Z_n, dove n è il prodotto di due numeri primi
- Naturalmente il numero deve essere un residuo quadratico modulo n (l'equivalente di un quadrato perfetto)
- Per ottenerlo, procediamo a rovescio: scegliamo a caso un numero x e lo eleviamo al quadrato

In [ ]:
```
p = 13
q = 23
n = p*q
print(n)
```

In [ ]:
```
x = randint(1,n-1)
y = x*x%n
print(y)
```

- Naturalmente, così facendo, una radice di y già la conosciamo, ed è ovviamente x. In realtà ne conosciamo subito un'altra, l'opposto di x in Z_n e cioè $n - x$

In [ ]:
```
(n-x)*(n-x)%n
```

In [ ]:
```
print(x,n-x)
```

- Poiché però n è il prodotto di due primi, le radici quadrate sono in tutto 4
- Come si calcolano le altre 2 (e, in generale, come si calcolano tutte e 4 le radici di un residuo quadratico y arbitrario)?

- Si noti innanzitutto che $y \bmod p$ e $y \bmod q$ sono a loro volta residui quadratici (modulo p e modulo q, rispettivamente)
- Infatti, una radice di $y \bmod p$ è $x \bmod p$:

$$((x \bmod p) \cdot (x \bmod p)) \bmod p = (x \cdot x) \bmod p = ((x \cdot x) \bmod n) \bmod p = y \bmod p$$

- Analogamente una radice di $y \bmod p$ è $x \bmod q$.
- Ora, il calcolo delle radici quadrate di y modulo n è facile proprio se conosciamo una radice di y modulo p e una modulo q
- Per scopi didattici (e per numeri piccoli) possiamo procedere col metodo brute force

In [ ]:
```
# Calcolo delle radici di y modulo p
yp = y%p
z1  = 1
while (z1*z1)%p != yp:
    z1 = z1+1
z2 = p-z1
print(z1,z2)
```

In [ ]:
```
# Calcolo delle radici di y modulo q
yq = y%q
w1  = 1
while (w1*w1)%q != yq:   # Se y è un r.q. mod n lo è anche mod q
    w1 = w1+1
w2 = q-w1
print(w1,w2)
```

- Per calcolare le 4 radici di y mod n calcoliamo preliminarmente le "solite" due quantità

$$c_1 = q \cdot (q^{-1} \bmod p)$$
$$c_2 = p \cdot (p^{-1} \bmod q)$$

In [ ]:
```
c1 = q*(modular_inverse(q,p))
c2 = p*(modular_inverse(p,q))
```

In [ ]:
```
print(c1%p,c1%q,c2%p,c2%q)
```

In [ ]:
```
R1 = (c1*z1+c2*w1)%n
R2 = (c1*z1-c2*w1)%n
R3 = (-c1*z1+c2*w1)%n
R4 = (-c1*z1-c2*w1)%n
print(R1,R2,R3,R4)
```

- Naturalmente, due delle quattro radici sono fra loro congrue modulo p e due sono congrue modulo q

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

In [ ]:
```
Euclid(R2-R3,n)==q
```

N.B. Prendere z_1 e w_1 è arbitrario

In [ ]:
```
(c1*z2+c2*w1)%n
```

In [ ]:
```
print(R1*R1%n)
print(R2*R2%n)
print(R3*R3%n)
print(R4*R4%n)
```

- Naturalmente due delle 4 radici coincidono con le due che (in questo caso) già conoscevamo

In [ ]:
```
print(x,n-x)
```

L'algoritmo generale

- Come per RSA, anche nel protocollo di Rabin la generazione delle chiavi richiede un solo parametro di input, che è la lunghezza in bit, indicata con N, dell'intero con cui si effettueranno le riduzioni modulari.
- Il messaggio M da cifrare dovrà essere un numero minore del modulo ovvero trasformato in un tale numero per mezzo di una funzione invertibile.

Generazione delle chiavi (Alice)

1. Genera due numeri primi a caso, $p \equiv 3 \pmod 4$ e $q \equiv 3 \pmod 4$, di lunghezza $N/2$ bit;
2. calcola il prodotto $n = p \cdot q$;
3. diffondi n come chiave pubblica e conserva la coppia p, q come chiave segreta.

- Nota: il fatto che i numeri p e q siano del tipo $4k + 3$ non è decisivo. Serve però a rendere più facile il calcolo delle radici quadrate in fase di decifrazione

Cifratura di un messaggio M (Bob)

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

Decifrazione del messaggio C (Alice)

- Ricordiamo preliminarmente che, se p divide n, allora vale $(x \bmod n) \bmod p = x \bmod p$. La dimostrazione è semplice e discende direttamente dalla definizione di modulo.

1. Calcola $M_p = C^{\frac{p+1}{4}} \bmod p$ e $M_q = C^{\frac{q+1}{4}} \bmod q$. Si noti quanto segue.
   - M_p è una delle radici quadrate di C modulo p perché

$$
\begin{aligned}
M_p^2 \bmod p &= \left(C^{\frac{p+1}{4}} \bmod p\right)^2 \bmod p \\
&= C^{\frac{p+1}{2}} \bmod p \\
&= \left(C^{\frac{p-1}{2}} \cdot C\right) \bmod p \\
&= \left((M^2 \bmod n)^{\frac{p-1}{2}} \cdot C\right) \bmod p \\
&= \left(\left((M^2 \bmod n) \bmod p\right)^{\frac{p-1}{2}} \cdot C\right) \bmod p \\
&= \left((M^2 \bmod p)^{\frac{p-1}{2}} \cdot C\right) \bmod p \\
&= (M^{p-1} \cdot C) \bmod p
\end{aligned}
$$

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

* Analogamente M_q è una delle radici quadrate di C modulo q

2. Calcola le seguenti quattro quantità, ciascuna delle quali è una radice quadrata di C modulo n
   - $M_1 = \left(\left(q \cdot (q^{-1} \bmod p)\right) M_p + \left(p \cdot (p^{-1} \bmod q)\right) M_q\right) \bmod n = (c_p M_p + c_q l$
   - $M_2 = n - M_1 = (n - M_1) \bmod n = (-M_1 \bmod n) = (-c_p M_p - c_q M_q) \bmod$
   - $M_3 = \left(\left(q \cdot (q^{-1} \bmod p)\right) M_p - \left(p \cdot (p^{-1} \bmod q)\right) M_q\right) \bmod n = (c_p M_p - c_q l$
   - $M_4 = n - M_3 = (-c_p M_p + c_q M_q) \bmod n$
3. Uno dei quattro valori è il messaggio originale

- Dimostriamo che si tratta effettivamente delle quattro radici di C modulo n
- Per quanto riguarda M_1 abbiamo

$$
\begin{aligned}
M_1^2 \bmod p &= \left(\left(\left(q \cdot (q^{-1} \bmod p)\right) M_p + \left(p \cdot (p^{-1} \bmod q)\right) M_q\right) \bmod n\right)^2 \bmod p \\
&= \left(\left(\left(q \cdot (q^{-1} \bmod p)\right) M_p + \left(p \cdot (p^{-1} \bmod q)\right) M_q\right) \bmod p\right)^2 \bmod p \\
&= (M_p \bmod p)^2 \bmod p \\
&= M_p^2 \bmod p \\
&= C \bmod p
\end{aligned}
$$

e, analogamente

$$M_1^2 \bmod q = C \bmod q$$

Il teorema cinese dei resti assicura quindi che

$$M_1^2 \bmod n = C \bmod n$$

- Per quanto riguarda M_2, banalmente

$$M_2^2 \bmod n = (n - M_1)^2 \bmod n = M_1^2 \bmod n = C \bmod n$$

- La dimostrazione per M_3 ed M_4 è identica

Un semplice esempio

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

In [ ]:
```
def rabinprime(N):
    '''Genera numeri primi n tali che n=4k+3'''
    while True:
        n = getprime(2**N)
        if n%4==3:
            return n
```

Generazione delle chiavi

In [ ]:
```
nbytes=16
nbits=nbytes*8
```

In [ ]:
```
p = rabinprime(nbits>>1); p
```

In [ ]:
```
q = rabinprime(nbits>>1); q
```

In [ ]:
```
n=p*q; n
```

In [ ]:
```
cp = q*modular_inverse(q,p)
cq = p*modular_inverse(p,q)
print(cp,cq)
```

Cifratura

In [ ]:
```
m=b'pippo'
```

In [ ]:
```
M = int.from_bytes(m,'big'); M
```

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

Decifrazione

In [ ]:
```
Mp = modexp(C,(p+1)>>2,p)
Mq = modexp(C,(q+1)>>2,q)
print(Mp,Mq)
```

In [ ]:
```
M1 = (cp*Mp+cq*Mq)%n
M2 = (cp*Mp-cq*Mq)%n
M3 = (-cp*Mp+cq*Mq)%n
M4 = (-cp*Mp-cq*Mq)%n
```

In [ ]:
```
# Per inciso...
Mi = (M1,M2,M3,M4)
[modexp(Mi[i],2,n) == C for i in range(4)]
```

In [ ]:
```
for i in range(4):
    print(Mi[i].to_bytes(16,'big'))
```

Equivalenza con il problema della fattorizzazione

- Dobbiamo dimostrare che se siamo in grado di decifrare (con un'incertezza di 1 su 4) allora possiamo anche fattorizzare n
- Il viceversa è ovvio

- Abbiamo visto che le quattro radici di $C \bmod n$ corrispondono alle quattro possibili combinazioni delle due radici modulo p con le due radici modulo q
  1. M_1 corrisponde alla coppia (M_p, M_q)
  2. M_2 corrisponde alla coppia $(-M_p \bmod p, -M_q \bmod q)$ = $(p - M_p, q - M_q)$
  3. M_3 corrisponde alla coppia $(M_p, -M_q \bmod q)$ = $(M_p, q - M_q)$
  4. M_4 corrisponde alla coppia $(-M_p \bmod p, M_q)$ = $(p - M_p, M_q)$
- Si consideri l'esempio di $n = 33 = 3 \cdot 11$ in cui 4 è ovviamente un residuo quadratico
- In tal caso abbiamo

$$M_p = 4^{\frac{3+1}{4}} \bmod 3 = 1 \quad \text{e} \quad M_q = 4^{\frac{11+1}{4}} \bmod 11 = 9$$

- Risulta inoltre

$$c_p = q \cdot (q^{-1} \bmod p) = 11 \cdot (11^{-1} \bmod 3) = 11 \cdot 2 = 22$$

e

$$c_q = p \cdot (p^{-1} \bmod q) = 3 \cdot (3^{-1} \bmod 11) = 3 \cdot 4 = 12$$

- Possiamo ora evidenziare bene la corrispondenza:
  1. Ritroviamo
     $M_1 = (c_p \cdot M_p + c_q \cdot M_q) \bmod n = (22 \cdot 1 + 12 \cdot 9) \bmod 33 = 130 \bmod 33 =$
  2. Poi $M_2 = (c_p \cdot (p - M_p) + c_q \cdot (q - M_q)) \bmod n = 68 \bmod 33 = 2$
  3. $M_3 = (c_p \cdot M_p + c_q \cdot (q - M_q)) \bmod n = 46 \bmod 33 = 13$
  4. $M_4 = (c_p \cdot (p - M_p) + c_q \cdot M_q) \bmod n = 152 \bmod 33 = 20$

In [ ]:
```
# verifica
modexp(31,2,33),modexp(2,2,33),modexp(13,2,33),modexp(20,2,33)
```

- Veniamo ora alla riduzione
- L'obiettivo è fattorizzare un numero n dato in input e l'ipotesi è di disporre di un algoritmo black-box che, dato un residuo quadratico C modulo n, restituisce una delle 4 possibili radici

di C modulo n

- La riduzione è un algoritmo tipo Las Vegas e funziona nel modo seguente
  1. Genero a caso un numero $r \in \mathbf{Z}_n, r \neq 0$
  2. Se $m = \mathrm{MCD}(r, n) \neq 1$ (la famosa botta di fortuna), restituisco m e n/m
  3. Altrimenti considero r come un "messaggio", calcolo $C = r^2 \bmod n$ e sottopongo C alla black box
  4. Se r' è il valore restituito dalla black-box, calcolo $m = \mathrm{MCD}(r - r', n)$
  5. Se $m > 1$ e $m \neq n$ restituisco i fattori m e n/m, altrimenti ritorno al passo 1

- La correttezza, ovvero il fatto che, con probabilità strettamente positiva, al passo 5 l'algoritmo trovi effettivamente un fattore di n, dipende proprio dalla corrispondenza delle radici con i residui modulo p e modulo q
- Sappiamo che il valore r' restituito dalla black box è una delle quattro radici di r^2 e che ciascuna di esse corrisponde a uno dei quattro modi con con cui possiamo combinare le due radici di r^2 modulo p con le due radici di r^2 modulo q: si tratta dei 4 valori
  - $r_p = r \bmod p$,
  - $-r_p = p - r_p$,
  - $r_q = r \bmod q$,
  - $-r_q = q - r_q$.

- Supponiamo dunque, fissare le idee, che la radice r da cui siamo partiti corrisponda alla coppia (r_p, r_q) e consideriamo le 4 possibili risposte dell'oracolo
  1. $r' = r$ (cioè anche r' corrisponde alla coppia (r_p, r_q)). In tal caso $r - r' = 0$ e $m = n$ e dunque, in questo caso, l'algoritmo non dà risposta;
  2. r' corrisponde alla coppia $(p - r_p, q - r_q)$. Allora

$$(r - r') \bmod p = r_p - (p - r_p) \bmod p = 2r_p \bmod p$$

  e analogamente

$$(r - r') \bmod q = 2r_q \bmod q$$

  da cui (sempre per il Teorema Cinese dei Resti) $r - r' \bmod n = 2r \bmod n$. Allora necessariamente $\mathrm{MCD}(r - r', n) = 1$ (altrimenti l'algoritmo si sarebbe fermato al passo 2) e pure in questo caso, l'algoritmo non può dare risposta 3. r' corrisponde alla coppia $(r_p, q - r_q)$. In questo caso abbiamo

$$(r - r') \bmod q = r_q - (q - r_q) \bmod q = 2r_q$$

  ma $(r - r') \bmod p = 0$ per cui $\mathrm{MCD}(r - r', n) = p$ e l'algoritmo termina con successo 4. r' corrisponde alla coppia $(p - r_p, r_q)$. Il caso è speculare al precedente e quindi l'algoritmo termina con successo
- Poiché r è scelto a caso, l'oracolo (fosse anche di tipo "malevolo") non può sapere a quale delle quattro radici corrisponde C e dunque con probabilità almeno 0.5 (in realtà un poco di più perché ci potrebbe essere il "colpo di fortuna" al passo 2) l'algoritmo termina un round con successo

In [ ]:
```
from random import choice
def oracle(p,q):
    '''Oracolo realizzato come coroutine'''
    n=p*q
    cp = q*modular_inverse(q,p)
    cq = p*modular_inverse(p,q)
    counter = 1
    C = yield "Oracle ready"
    while True:
        rp = modexp(C,(p+1)>>2,p)
        rq = modexp(C,(q+1)>>2,q)
        r1 = (cp*rp+cq*rq)%n
        r2 = n-r1
        r3 = (cp*rp-cq*rq)%n
        r4 = n-r3
        print(f"Invocazione {counter} dell'oracolo")
        counter += 1
        C = yield choice([r1,r2,r3,r4])
```

In [ ]:
```
from random import randint
from ACLIB.utils import Euclid
```

In [ ]:
```
N = 30
p = rabinprime(N)
q = rabinprime(N)
n = p*q
print(f"Il numero da fattorizzare è {n} e i fattori sono {p} e {q}")
O = oracle(p,q)     # creazione dell'oracolo
print(O.send(None)) # attivazione dell'oracolo, che "ora" è fermo a riga
while True:
    r = randint(1,n-1)
    R = (r*r)%n
    print(f"Il numero casuale scelto è {r}")
    rp = O.send(R)  # l'oracolo riceve input, calcola, restituisce e si f
    m = Euclid(r-rp,n)
    if m==n:
        print(f"L'oracolo ha restituito ancora rp = r = {r}")
    elif  m==1:
        print(f"L'oracolo ha restituito rp = n-r = {rp} ")
    else:
        print("L'oracolo ha restituito una radice utile per la fattorizza
        break
    r = randint(1,n-1)
    R = (r*r)%n
# riveliamo ora p e q
print(f"I fattori calcolati sono {m} e {n//m}")
```

In [ ]:
```
from ACLIB.utils import modular_inverse
```

Il sistema crittografico a chiave pubblica di ElGamal

- Si tratta di un sistema proposto da Taher ElGamal nel 1985, quindi non è il primo crittosistema a chiave pubblica.
- Esso segue di qualche anno il sistema RSA, brevettato nel 1977.
- Lo trattiamo prima dell'RSA, e subito dopo il protocollo di DH per lo scambio i chiavi, in quanto è strettamente legato a quest'ultimo.
- Poiché non è stato brevettato, per molto tempo è stato inserito, insieme al protocollo DSA (Digital Signature Algorithm), in diverse suite crittografiche, ad esempio GNUPG.

Descrizione del protocollo

- Come in qualsiasi protocollo crittografico a chiave pubblica, ci sono due momenti distinti:
  1. generazione della coppia di chiavi, pubblica e privata, con la conseguente "diffusione" (pubblicazione, appunto) della prima;
  2. esecuzione di comunicazioni cifrate fra le due parti, processo a sua volta composto da cifratura e decifrazione.
- Si noti che il protocollo è asimmetrico e dunque, per fissare le idee, considereremo la generazione delle chiavi dalla sola parte di Alice.

Generazione delle chiavi (Alice)

1. Alice determina i parametri del protocollo:
   - un numero primo p di lunghezza appropriata,
   - una radice primitiva g di $\mathbf{Z}_p^*$,
   - il valore $A = g^a \bmod p$, dove $a \in \mathbf{Z}_p^*$ è un numero scelto uniformemente a caso.
2. Alice conserva a come propria chiave segreta e provvede alla diffusione della terna (p, g, A) come corrispondente chiave pubblica.

- Per questa parte del protocollo, si può osservare che la differenza con DH è solo nei destinatari della comunicazione (non solo Bob bensì tutti coloro che dovranno inviare messaggi cifrati ad Alice).

Cifratura di un messaggio M (Bob)

1. Bob recupera la chiave pubblica di Alice: (p, g, A);
2. sceglie un numero $b \in \mathbf{Z}_p^*$ uniformemente a caso;
3. calcola le due quantità $B = g^b \bmod p$ e $c = \left(A^b \cdot M\right) \bmod p$;
4. invia ad Alice la coppia $C = (B, c)$, che costituisce il messaggio cifrato.

Decifrazione del messaggio $C = (B, c)$ (Alice)

1. Alice calcola la quantità $Z = B^a \bmod p$;
2. calcola (usando l'algoritmo di Euclide esteso) $Z^{-1} \bmod p$;
3. calcola $M = \left(Z^{-1} \cdot c\right) \bmod p$.

- Si noti che il messaggio deve essere interpretabile come numero minore di p.

Correttezza

- È un'immediata conseguenza del fatto che $Z = B^a \bmod p = A^b \bmod p$ esattamente come nel protocollo di Diffie-Hellman

Efficienza

- A parte la pre-computazione delle chiavi (operazione eseguita una sola volta), la complessità della cifratura è dominata da due calcoli di potenze modulari (cui va aggiunto un prodotto).
- La decifrazione richiede invece il calcolo di una potenza, di un inverso e di una moltiplicazione.
- Si tratta di poche operazioni che però sono eseguite su numeri di oltre 1000 bit.
- Se il messaggio fosse più lungo della lunghezza del modulo p, Bob dovrebbe spezzarlo in blocchi e ripetere la cifratura, usando però un valore k differente per ogni singolo blocco (vedremo subito perché).
- Con molti blocchi, e cioè con messaggi lunghi, cifratura e decifrazione "asimmetriche" diventano in realtà processi alquanto onerosi, soprattutto se paragonati all'efficienza di moderni algoritmi di cifratura simmetrica (che possono anche avere supporto hardware).
- Per questa ragione, la crittografia asimmetrica viene usata congiuntamente, e in modo sinergico, a quella simmetrica.
- In particolare, protocolli asimmetrici vengono utilizzato in fase di autenticazione delle parti e per lo scambio di chiavi (come già abbiamo osservato).
- Per la comunicazione vera e propria si utilizza invece un protocollo simmetrico.

Sicurezza

- Il problema di violare lo schema di cifratura di ElGamal, cioè mettere in chiaro il messaggio M a partire dai valori p, g, A, b e c, è equivalente a risolvere il CDH problem (vedi cella succesiva).
- Se quindi vale l'ipotesi CDH, il crittosistema di ElGamal è sicuro.
- Un errore da non commettere è di utilizzare per cifrature differenti lo stesso valore b (randomicamente scelto solo la prima volta).
- Se infatti si usa lo stesso valore di b, anche B non cambia e dunque, a due messaggi distinti M_1 ed M_2, corrisponderebbero le due cifrature:

$$C_1 = \left(B, c_1 = \left(A^b \cdot M_1\right) \mod p\right) \qquad \text{e} \qquad C_2 = \left(B, c_2 = \left(A^b \cdot M_2\right) \mod p\right)$$

- Moltiplicando, ad esempio, $c_1^{-1} \mod p$ per c_2, si vede che:

$$c_1^{-1} \cdot c_2 \bmod p = (M_1^{-1} \cdot M_2) \bmod p$$

ovvero che

$$M_2 = \left(M_1 \cdot \left(c_2 \cdot c_1^{-1}\right)\right) \mod p$$

Se dunque Eva fosse in grado (per qualsiasi motivo) di decifrare il messaggio M_1, potrebbe decifrare anche i successivi messaggi cifrati con lo stesso valore b.

$CDH \Rightarrow$ ElGamal è sicuro

- Usiamo l'implicazione contropositiva: supponiamo cioè di poter "rompere" ElGamal e siamo in grado di risolvere l'assuzione CDH
- La dimostrazione è immediata nel momento in cui riflettiamo sul fatto che ElGamal "oscura" il messaggio moltiplicandolo (modulo p) proprio per una quantità $A^b \bmod p$, che corrisponde al segreto condiviso di Diffie-Hellman.
- Se dunque siamo in grado di mettere in chiaro M (senza il calcolo diretto del logaritmo discreto $a = \log_g A$), possiamo risalire alla quantità $A^b \bmod p = g^{a \cdot b} \bmod p$, proprio la quantità che la CDH assumption chiede di calcolare.

Qualche esperimento con la libreria Crypto di Python

- Cifratura e firma digitale non sono più supportate in PyCryptodome
- Rimane disponibile la creazione di chiavi
- Come esercizio, forniamo qui una "textbook version" di encrypt e decrypt

In [ ]:
```
from Crypto.PublicKey import ElGamal
from Crypto.Random.random import Random
from Crypto.Hash import SHA
from Crypto.Math import Numbers
```

In [ ]:
```
key = ElGamal.generate(256, Random.get_random_bytes)
```

In [ ]:
```
key.__dict__
```

- x e y sono le quantità che, nella descrizione del protocollo, abbiamo indicato con a e A

In [ ]:
```
class EGKey(ElGamal.ElGamalKey):
    '''Define ElGamal key with "textbook" encryption/decryption implement
    def __init__(self,l,randfun=Random.get_random_bytes):
        '''The super().__init__ method would be useless here. pycryptodom
           function to generate ElGamal keys. We generate one and copy
           the dictionary to self.
           Simply returning the generated key would give an ElGamal key
           and not an EGKey'''
        self.__dict__ = ElGamal.generate(l,randfun).__dict__
    def publickey(self):
        '''Returns an EGKey public key corresponding to self
           (i.e. without the secret information)
        '''
        pubkey = super().__new__(EGKey)   # create an "empty" ElGamalKey
        pubkey.p = self.p
        pubkey.g = self.g
        pubkey.y = self.y
        return pubkey
    def decrypt(self,ciphertext):
        '''Decrypt cipertext using self key'''
        B = Numbers.Integer(ciphertext[0])
        c = Numbers.Integer(ciphertext[1])
        B.inplace_pow(self.x,self.p)
        M = (B.inverse(self.p)*c)%self.p
        return M.to_bytes(self.p.size_in_bytes())
    def encrypt(self,plaintext):
        '''Encrypt plaintext using self key'''
        assert len(plaintext) <= self.p.size_in_bytes()
        while (b:=Numbers.Integer.random_range(min_inclusive=1,max_inclus
               and b.gcd(self.p-1)!=1:
            pass
        g = Numbers.Integer(self.g)
        A = Numbers.Integer(self.y)
        M = Numbers.Integer.from_bytes(plaintext)
        A.inplace_pow(b,self.p)
        return g.inplace_pow(b,self.p),(A*M)%self.p
```

In [ ]:
```
mykey = EGKey(256)
```

In [ ]:
```
mykey.__dict__
```

In [ ]:
```
pubkey = mykey.publickey()
```

In [ ]:
```
pubkey.__dict__
```

In [ ]:
```
message = b"Quite a short text"
```

In [ ]:
```
ciphertext = pubkey.encrypt(message);ciphertext
```

In [ ]:
```
mykey.decrypt(ciphertext)
```

Esercizio

- Modificare le funzioni di cifratura e decifrazione in modo da evitare i *byte di padding* nel testo rimesso in chiaro