Esercizi di approfondimento

Common modulus failure

Un utente A possiede due coppie di chiavi pubbliche-private RSA, relative allo stesso modulo n dato da

$$n = 825500608838866132701444300844117841826444264266030066831623$$

Le due chiavi pubbliche sono $K_1^+ = <3, n>$ e $K_2^+ = <11, n>$. Un secondo utente invia ad A, in tempi diversi, lo stesso messaggio m, cifrato prima con la chiave K1 e poi con la chiave K2 . Un attaccante intercetta i relativi plaintext, $c_1 = E_{K_1^+}[m]$ e $c_2 = E_{K_2^+}[m]$, che numericamente valgono

$$c_1 = 415459980059712388764580516278528357540868548132004893964433$$

$$c_2 = 884141165346707443294744910953393011210663087557694028365577$$

Ricavare m a partire dalle informazioni disponibili, senza fattorizzare n o ricavare gli esponenti privati.

Soluzione

Sapendo che $c_1 = m^{e_1} \mod n$ e $c_2 = m^{e_2} \mod n$ posso moltiplicarli tra di loro ottenendo:

$$c_1 * c_2 = m^{e_1} * m^{e_2} = m^{e_1+e_2} \mod n$$

Se $e_1 + e_2 = 1$ allora ho trovato m.

Se $MCD(e_1, e_2) = 1$, cioè se e_1 e e_2 sono coprimi, Posso esprimere $xe_1 + ye_2 = 1$.

Il problema precedente diventerà:

$$c_1^x * c_2^y = (m^{e_1})^x * (m^{e_2})^y = m^{xe_1+ye_2} = m^1 = m \mod n$$

Effettivamente 3 e 5 sono coprimi e posso trovare x e y tramite l'algoritmo di Euclide esteso:

$$11 = 3 * 3 + 2$$

$$3 = 1 + 2 + 1$$

$$1 = 3 - 1 + 2 = 3 - 1 * (11 - 3) = -1 * 11 + 4 + 3$$

La x è negativa, ma sapendo che $-1 \equiv 11^{-1} \mod 3$, posso scrivere $-1 + 3 = 2$, quindi ho:

$$x = 2$$

$$y = 4$$

Sostituendo i valori si ottiene: $c_1^4 * c_2^2$ (sono stati invertiti i valori per rispettare i relativi inversi).

Effettuando i calcoli otteniamo:

$$\begin{aligned} &415459980059712388764580516278528357540868548132004893964334^4 \\ &* 884141165346707443294744910953393011210663087557694028365772^2 \\ &\quad \bmod 825500608838866132701444300844117841826444264266030066831623 \\ &= 564140501104607297831987135512845214089854084977820388226740 \end{aligned}$$

Timing attack contro esponenziazione modulare (Kocher 1996)

Domanda a

$Var(X) \stackrel{def}{=} E[(X-\mu)^2] = E[X^2] - \mu^2$

Se X e Y sono variabili aleatorie indipendenti, allora $Var[X+Y] = Var[X] + Var[Y]$, infatti:

$$\begin{aligned} Var[X+Y] &= E[(X-\mu_1)^2 + (Y-\mu_2)^2] \\ &= E[(X-\mu_1)^2] + E[(Y-\mu_2)^2] \quad \text{(Per la linearità del valore atteso)} \\ &= Var[X] + Var[Y] \end{aligned}$$

Domanda b

$$\begin{aligned} T - T' = &\underbrace{(T_i - T_i')}_{\text{Tempo comune}} \\ + &\underbrace{\sum_{j=i-1}^{0} T_j}_{\text{Tempo differente}} \end{aligned}$$

Domanda c

$$\begin{cases} Var\left[T - T'\right] = i\nu & \text{se } d_i = d' \\ Var\left[T - T'\right] = \underbrace{2^\nu}_{(T_i - T_i')} \quad \underbrace{i\nu}_{\sum_{j=i-1}^{0} T_j} & \text{se } d_i \neq d' \end{cases}$$

Domanda d

Facendo riferimento all'equazione della domanda C, possiamo stabilire che abbiamo indovinato il bit giusto se ha associato una varianza minore, infatti se $d_i = d'$ allora $Var\left[T - T'\right] = i\nu$, contro $2^\nu i\nu$.

Domanda f

L'attacco è left to right e si parte ponendo $d_{k-1} = 1$, si indovinano i vari bit scegliendo quello con varianza minore e ci si sposta verso destra fino a d_0. ### Domanda g Non possiamo usare la media perchè non si somma coma la varianza.

Esercizi di programmazione

Timing Attack

Secondo la teoria possiamo effettuare un Timing attack e scoprire i bit dell' esponente in maniera iterativa nel seguente modo: 1. Partendo dal bit a sinistra posto a 1, calcolare la varianza tra tante osservazioni aggiungendo prima uno 0 e poi un 1 2. Selezionare il bit che porta alla varianza minima

Avendo l'esponente di 64 bit procediamo per altre 63 volte come segue:

```
def main():
    ta = TimingAttack()
    exponent = [1]
    for _ in range(1,64):
        variance = generateObservations(exponent,ta)
        if variance[0] < variance[1]:
            exponent.append(0)
        else:
            exponent.append(1)
```

Le osservazioni vengono generate tramite la funzione `generateObservations(exponent,ta)` che prende in input la lista dell'esponente attualmente trovata e l'oggetto ta che simula ad esempio una smart card rubata. Come output restituisce una lista delle due varianze calcolate: - varianza con il bit 0 in posizione 0 - varianza con il bit 1 in posizione 1

```
def generateObservations(exponent:list[int],ta:TimingAttack)->list[int]:
    observations0 = []
    observations1 = []
    for _ in range(2000):
        chipertext = np.random.randint(0,(2**62-1))
        realTime = ta.victimdevice(chipertext)
        observations0.append(trybit(0,ta,exponent,realTime,chipertext))
        observations1.append(trybit(1,ta,exponent,realTime,chipertext))
    var0 = np.array(observations0)
    var1 = np.array(observations1)
    return [np.var(var0),np.var(var1)]
```

Per 2000 osservazioni si genera un ciphertext casuale e si calcola il tempo che la smart cart impiega a decifrare con `ta.victimdevice(chipertext)`. Adesso dobbiamo calcolarci i tempi della decifratura per le prime k iterazioni, con k la dimensione attuale dell'esponente che abbiamo trovato. Effettuando la differenza tra il ciphertext e il tempo appena calcolato (prima con 0 e poi con 1) tramite la funzione `trybit(0,ta,exponent,realTime,chipertext)`, abbiamo generato un'osservazione (una con il bit 0 e una con il bit 1).

A fine ciclo possiamo calcolare la varianza di questi due array e restituire il valore al main.

```
def trybit(bit:int,ta:TimingAttack,exponent:int,realTime:int,chipertext:int)->int:
    exponent.append(bit)
    observation = realTime - ta.attackerdevice(chipertext,exponent)
    del exponent[-1]
    return observation
```

La funzione `trybit` calcola semplicemente la differenza di tempo impiegata nella decifratura tra la vera smart cart e il ciclo troncato per l'attacco.

Implementazione di algoritmi per crittografia a chiave pubblica

Algoritmo di Euclide esteso

Implementato in `ExtendedEuclideanAlgorithm.py`

L'algoritmo prende in input due numeri interi a e b e restituisce una `tupla` contenente `(mcd,x,y)` dove: - *mcd* è il massico comun divisore tra a e b - x è l'inveso moltiplicativo di a - y è l'inveso moltiplicativo di a

```
def ExtendedEuclideanAlgorithm(a:int, b:int) -> tuple[int, int, int]:
    x,y, u,v = 0,1, 1,0
    while a != 0:
        q = b//a
        r = b%a
        m, n = x-u*q, y-v*q
        b,a = a,r
        x,y = u,v
        u,v = m,n
    mcd = b
    return mcd, x, y
```

Algoritmo di esponenziazione modulare veloce

Implementato in `FastExponential.py`

Per gestire esponenti molto grandi si prende la codifica binaria sotto forma di stringa. Pardendo da sinistra, si effettua l'operazione `result = (result * result) % modulus` e se il bit dell'esponente è `1` effettuiamo anche `result = (result * base) % modulus`. L'algoritmo termina una volta scanditi tutti i bit dell'esponente.

```
def fast_modular_exponentiation(base:int, exponent:int, modulus:int)->int:
    exponent = bin(exponent)[2:]
    result = 1
    for i in range(0,len(exponent)):
        result = (result * result) % modulus
        if exponent[i] == '1':
```

```
            result = (result * base) % modulus
    return result
```

Test di Miller-Rabin

Implementato in `PrimalityTest.py`

```
from FastExponential import fast_modular_exponentiation
def rabin_test(n:int)->bool:
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    m = n - 1
    k = 0
    while m % 2 == 0:
        m //= 2
        k += 1
    a = 2
    b = fast_modular_exponentiation(a, m, n)
    if b == 1 or b == n-1:
        return True
    for _ in range(k-1):
        b = fast_modular_exponentiation(b, 2, n)
        if b == n-1:
            return True
    return False
```

Algoritmo per la generazione di numeri primi

Implementato in `PrimeGenerator.py`

L'algoritmo genera casualmente un numero molto grande, di default è a 1024 bit fino a che non supera il test di primalità.

```
from PrimalityTest import rabin_test
import random

def prime_generator(bit_lenght:int = 1024)->int:
    while True:
        n = random.randrange(2**(bit_lenght-1), 2**bit_lenght)
        if rabin_test(n):
            return n
```

Schema RSA, con e senza ottimizzazione CRT

Implementato in `RSA.py`

Vengono utilizzate le seguenti librerie:

```python
from PrimeGenerator import prime_generator
import numpy as np
import time
```

Il primo passo è quello di generare una chiave, sia pubblica che privata, restituite come tupla

```python
def rsa_keygen(bit_lenght: int = 1024) -> tuple:
    p = prime_generator(bit_lenght)
    q = prime_generator(bit_lenght)
    n = p * q
    phi = (p - 1) * (q - 1)
    e = 65537
    d = pow(e, -1, phi)
    return (e, n), (d, n),(p,q)
```

Vengono anche restituita la tupla (p,q) contenente i due numeri primi generati, per essere usati dalla funzione di decifratura con il CRT.

Le funzioni di cifratura e decifratura sono semplicemente:

```python
def crypt(m: int, e: int, n: int) -> int:
    return pow(m, e, n)

def decrypt(c: int, d: int, n: int) -> int:
    start = time.time()
    return pow(c, d, n),time.time()-start
```

cioè:

$$ciphertext = m^d \mod n$$

$$plaintext = c^d \mod n$$

La decifratura con il CRT è la seguente:

```python
def decrypt_crt(c:int,d:int,p:int,q:int) -> int:
    Sp = d % (p - 1)
    Sq = d % (q - 1)
    start = time.time()
    q_inv = pow(q, -1, p)
    #p_inv = pow(p, -1, q)
    m1 = pow(c, Sp, p)
    m2 = pow(c, Sq, q)
    h = (q_inv * (m1 - m2)) % p
    return m2 + h * q,time.time()-start
```

È possibile usare lo speed-up solo per la decifratura perchè bisogna conoscere p e q, ovvero la fattorizzazione di n che conosce soltanto chi ha la chiave privata.

Confrontare le prestazioni, in termini di tempo di esecuzione, delle versione senza e con CRT

Eseguendo il `main()` del modulo `RSA.py`:

```python
def main():
    base = []
    crt = []
    public, private,primes = rsa_keygen()
    for _ in range(100):
        m = np.random.randint(2**61, 2**62)
        c = crypt(m, public[0], public[1])
        m,time_base = decrypt(c, private[0], private[1])
        m,time_crt = decrypt_crt(c, private[0], primes[0],primes[1])
        base.append(time_base)
        crt.append(time_crt)

    print("Base: ",np.mean(base))
    print("CRT:  ",np.mean(crt))
    print("Percentuale di miglioramento: ",\
          ((np.mean(base)-np.mean(crt))/np.mean(base))*100,"%")
```

otteniamo il seguente output: (i risultati possono variare leggermente)

```
Base:  0.02277808666229248
CRT:   0.006162500381469727
Percentuale di miglioramento:  72.9454871568677 %
```