Set 2 di esercizi

MICHAEL CAVICCHIOLI

7149344

Anno Accademico 2023-2024

ESERCIZI DI APPROFONDIMENTO

2.1 PROBABILITÀ DI ERRORE IN MILLER-RABIN

(a): si vuole dimostrare che (xy) mod n è un testimone di Fermat di n in Z_n^*, cioè che $(xy, n) = 1$ e che $(xy)^{n-1} \not\equiv_n 1$.
Da ciò segue che $(x)^{n-1}(y)^{n-1} \not\equiv_n 1$ e applicando l'ipotesi che y non è un testimone di Fermat, quindi $(y)^{n-1} \equiv_n 1$, si ha che $(x)^{n-1} * 1 \not\equiv_n 1$, e che quindi, quest'ultima parte, sarà uguale ad un qualche k, con k intero diverso da 1.
Pertanto, si è dimostrato che (xy) mod n è un testimone di Fermat di n in Z_n^*.

(b): si vuole dimostrare che dati x testimone di Fermat ed y,y'$\in Z_n^*$ due diversi non testimoni di Fermat, valga: $xy \not\equiv_n xy'$.
Si suppone per assurdo che valga la seguente congruenza:
$xy \equiv_n xy'$.
Si osserva che x è un testimone di Fermat, pertanto si ha che $(x, n) = 1$, quindi $\exists!\ x^{-1} \in Z_n^*$.
Questo implica che da $xy \equiv_n xy'$ si ottiene $y \equiv_n y'$, ma questo è un assurdo, poichè per ipotesi si ha che y,y' sono due diversi non testimoni di Fermat.
Pertanto, si è dimostrato che vale la non congruenza.

(c): si vuole concludere, da (a) e (b), che se c'è almeno un testimone di Fermat (n non è Carmichael) $\implies$ in Z_n^*, per ciascuno dei non testimoni, c'è almeno un testimone distinto.
È noto che $\exists$ almeno un testimone x perchè n non è Carmichael $\implies \forall$ y non testimone si ha un testimone dal prodotto xy, come spiegato in (a).
Inoltre, si sa che $\forall$ y'$\neq$ y, si ha che $xy \neq xy'$, come spiegato in (b) $\implies$ i testimoni per y e y' sono due testimoni distinti tra loro.
Pertanto, si ha almeno un testimone per ogni non testimone e quindi si hanno almeno $\frac{|Z_n^*|}{2} = \frac{\phi(n)}{2}$.

(d): si vuole concludere che se n è composto dispari non Carmichael, la probabilità che l'algoritmo ritorni *vero* è almeno $\frac{1}{2}$.

Si osserva che se n è composto dispari non Carmichael, per quanto detto in precedenza, n avrà $\frac{\phi(n)}{2}$ testimoni di Fermat.
Inoltre, l'algoritmo di Fermat prevede di scegliere un $x \in (1, n-1)$ e di controllare se il massimo comun divisore con n è ugale o diverso da 1.
Se l'MCD risulta essere uguale a 1, allora si applica l'algoritmo di Fermat per vedere se vale $x^{n-1} \not\equiv_n 1$. Se questo vale, allora x è un testimone di Fermat e la probabilità di trovarlo è di almeno $\frac{1}{2}$, per quanto detto in precedenza, pertanto si ritorna *vero*, cioè n composto.
Altrimenti (MCD $\neq 1$), si ritorna *vero*, perchè questo significa che x e n hanno fattori in comuni, quindi n composto.

2.4 COMMON MODULUS FAILURE

La soluzione di tale esercizio è stata implementata nel file *common_modulus_failure.py* e per eseguirlo basta richiamare il suo nome, preceduto da *python* o *python3*, nella cartella dove risiede il file, oppure aprirlo con Visual Studio Code ed eseguirlo da lì (per fare questo, bisogna aver installato l'estensione di Python per tale editor).
In prima istanza si è andati a inizializzare le variabili di cui erano note le informazioni (n, c_i, e_i).
Dopodichè, si è andati a richiamare l'algoritmo di *Euclide esteso* per trovare, mediante l'identità di Bèzout, i due esponenti da dare ai rispettivi ciphertext e controllare, mediante delle semplici *print*, quale dei due fosse l'esponente negativo.
In seguito, richiamando il medesimo algoritmo, si è andati a calcolare l'inverso (modulo n) del primo chipertext, poichè a lui corrispondeva l'esponente negativo.
Infine, si è andati a calcolare il valore del messaggio applicando la seguente formula:
$(((c1)^x \mod n) * ((c2)^y \mod n)) \mod n$, per poi stamparlo a schermo con una *print*.
Il valore di tale m risulta essere pari a 65535.
Si fa notare che il valore di c1 corrisponde al c1 inverso che si è andati a calcolare in precedenza e il valore dell'esponente x è stato preso in valore assoluto, questo perchè, per esempio:
$4^{-5} = (4^{-1})^5$.

ESERCIZI DI PROGRAMMAZIONE

3.1 IMPLEMENTAZIONE DI ALGORITMI PER CRITTOGRAFIA A CHIAVE PUBBLICA

La soluzione di questo esercizio è implementata nei due file Python *algoritmi.py* e *main.py*, dove nel primo sono riportati tutti gli algoritmi e funzioni di aiuto, mentre nel secondo sono riportati tutti i richiami di tali algoritmi.
Per eseguire i codici bisogna scrivere *python main.py*, o *python3 main.py*, nella cartella in cui risiede tale file oppure eseguirlo da Visual Studio Code (ma bisogna aver installato l'estensione Python per tale editor).

1. **Algoritmo di Euclide esteso**: questo algoritmo permette di calcolare l'MCD di due valori a e b e, se essi sono coprimi, anche i loro inversi (identità di Bèzout), dove la formula è la seguente: $ax + by = 1$.
   Il metodo Python che si è andati a creare è *euclide_esteso*.
   Questo metodo è ricorsivo e calcola i resti delle divisioni fino a quando il primo valore è uguale a 0 (caso base). Una volta finita la ricorsione, quindi procedendo a ritroso nelle chiamate presenti nello stack, si aggiornano i valori delle variabili, andando a calcolare quelle che sono le identità di Bèzout. Una volta finite le chiamate presenti nello stack, viene ritornato l'MCD e le due identità calcolate.

2. **Algoritmo di esponenziazione modulare veloce**: questo algoritmo permette di calcolare, molto velocemente, moduli di valori interi con esponenti molto grossi.
   Il metodo Python che si è creato è *esponenziazione_modulare_veloce*.
   Questo metodo prende in input la base, l'esponente e il modulo, controlla che gli input siano passati correttamente e successivamente calcola la rappresentazione binaria dell'esponente.
   Dopodichè, cicla dal bit più significativo (quello più a sinistra) fino a quello meno significativo e ad ogni iterazione calcola quello che sarà il risultato finale, come il quadrato di sè stesso in modulo quello passato come input e raddoppia la variabile contatore.
   In seguito, viene controllato che il bit i-esimo sia uguale ad 1 e in caso positivo, quello che sarà il risultato finale viene moltiplicato

per la base passata in input, poi applicato il medesimo modulo e incrementato il contatore di 1.

3. **Test di Miller-Rabin**: è uno dei test di primalità più famosi, il quale usa, al suo interno, il *piccolo teorema di Fermat*, ma rafforzandolo con la proprietà dei *resti quadratici*, ovvero che dato n primo e $0 < x < n \implies x^2 \equiv_n 1 \leftrightarrow x \equiv_n \pm 1$.
   Il test di Miller-Rabin controlla che un numero intero sia composto o meno. Se il test restituisce *vero*, allora il numero è sicuramente composto, altrimenti non è detto che il numero sia primo, infatti esiste una probabilità di errore che è al più di $\frac{1}{2}$, ma può essere ridotta ad al più di $\frac{1}{4}$.
   La prima parte del test consiste nel prendere due numeri interi n e x, e riscrivere n-1 come $x^r m$, con $r \geqslant 1$ e $m > 0$. Così facendo, si è andati a calcolare il primo termine della sequenza, $x_0 \equiv_n x^m \mod n$, e controllare che non sia un fattore banale.
   Successivamente, si è andati a calcolare, e controllare, che i restanti termini della sequenza, definiti come $x_i = x_{i-1}^2 \mod n$, fossero fattori non banali, perchè, in caso positivo, essi potrebbero essere numeri non composti.
   Se il ciclo finisce, si restituisce *vero*, poichè n è composto.
   Il metodo Python che riporta il seguente test è *miller_rabin*.

4. **Algoritmo per la generazione di numeri primi**: algoritmo che prende in ingresso un numero k di bit e crea un vettore di k valori randomici binari, dove quelli più e meno significativi sono posti a 1 (più significativo uguale 1 implica di considerare numeri interi dell'ordine di 2^k, mentre il meno significativo uguale a 1 implica di considerare solo i numeri dispari, poichè quelli pari sono composti). Dopodichè, si è andati a convertire il vettore ad un numero intero n e generato, in maniera randomica, un valore x compreso tra 2 e n-1. Sapendo quanto detto al punto precedente, si è controllato che il numero n non sia composto, tramite il test di Miller-Rabin, e che l'MCD(n, x) sia uguale ad 1. Avendo impostato la tolleranza a $\frac{1}{4}$, e un valore di tolleranza soglia molto basso, si è andati ad aggiornare la tolleranza, ogni volta che il numero sembrava non essere composto, generare un nuovo x per controllare che n fosse effettivamente primo e restituire il numero n solo quando la tolleranza sarebbe stata minore o uguale a quella di soglia.
   Il metodo Python che raffigura questo algoritmo è *generatore_numeri_primi*.

5. **Schema RSA, con e senza ottimizzazione CRT**: come primo passo viene richiamato il metodo *genera_valori_rsa*[1], il quale serve per calcolare i parametri di RSA: n, p, q, $\phi(n)$, e, d.
   Successivamente, vengono generati, uno alla volta, 100 plaintext di lunghezza 1000, i quali vengono passati alla funzione di encryption per ottenere i relativi ciphertext.
   Dopodichè, viene preso il tempo per eseguire la decryption classica di RSA, vengono pre computati i valori per la decryption tramite CRT e preso il tempo per eseguire quest'ultima.
   Infine, vengono stampati a schermo i tempi di ogni decryption e quante volte una decryption ha richiesto più tempo dell'altra.
   Il metodo Python che implementa tale codice è *rsa*.

3.2 ATTACCO AL DECRYPTION EXPONENT IN RSA

(a): si vuole spiegare l'esistenza certa dell'indice j e del perchè, quando termina, l'algoritmo fornisce un fattore non banale di n.
Avendo posto ed - 1 $= 2^r * m$ e osservando che il teorema di Eulero afferma che dato un $x \in \mathbb{N}$, se (x, n) = 1 $\implies x^{\phi(n)} \equiv_n 1$, si è certi dell'esistenza di j poichè, riscrivendo l'ultimo passo dell'algoritmo, si ottiene che:
$x_r = x^{2^r * m} \mod n = x^{ed-1} \mod n \equiv_n x^{(ed-1) \mod \phi(n)} = x^0 = 1$.
Questo significa che, proseguendo con la sequenza degli $x^{2^i * m} \mod n, \forall i = 0, \ldots, r$, si incontrerà un valore diverso da 1, il quale elevato al quadrato, in modulo n, sarà uguale a 1.
Quindi, si saprà che $x_{j-1} - 1, x_{j-1} + 1$ avranno fattori distinti in comune con n $\implies n \mid (x_{j-1} - 1) * (x_{j-1} + 1)$ e non solo uno dei due, perchè, se così fosse $\implies x_{j-1} \equiv_n \pm 1$, che è assurdo per ipotesi dell'algoritmo.
Pertanto, i due termini hanno un fattore non banale in comune con n, che è possibile ricavare mediante l'MCD tra il primo fattore e n e il secondo fattore e n.

(b): il metodo Python che contiene il codice è *rsa_3_2*.
In primo luogo vengono definiti due vettori, uno per il tempo di

1 In primo luogo genera due primi, p e q, di 200 bit, poi calcola n e $\phi(n)$. Successivamente, genera l'esponente segreto d dell'ordine di grandezza di n e calcola e, l'esponente pubblico, come l'identità di Bèzout che, moltiplicata per d in modulo $\phi(n)$ è congruo a 1. Dopodichè, se e risulta negativo, allora calcola il modulo per trovare il valore positivo ed infine, se e è maggiore di d, allora scambia i valori, poichè l'attaccante potrebbe trovare d per tentativi se troppo piccolo.

esecuzione dell'algoritmo, mentre il secondo per il numero di iterazioni totali.
Dopodichè, viene creato un ciclo di 100 iterazioni per testare il metodo *decryptionexp*, definito nel testo dell'esercizio. All'interno del ciclo, vengono generati i parametri di RSA, utili per questo fine, viene preso il tempo della decryption e salvati sia il numero di iterazioni compiuto dall'algoritmo, che il tempo della sua esecuzione, per poi stampare a schermo quanto richiesto, cioè: il numero medio di iterazioni, il tempo medio di esecuzione e la varianza.
Nel metodo *decryptionexp*, come primo passo, vengono trovati i valori di r ed m, tali per cui è possibile scrivere $ed - 1 = 2^r m$, con $r \geqslant 1$ e m dispari.
In secondo luogo, all'interno di un ciclo while, viene generato un numero x casuale, compreso tra 2 e n-1, e controllato che l'MCD con n sia diverso da 1: in questo caso, n è composto e quindi l'algoritmo termina restituendo il numero di iterazioni compiute.
In caso contrario, viene incrementato il numero di iterazioni e viene utilizzata una 'variante' del test di Miller-Rabin, definita dal metodo *miller_rabin_modificato*, che esegue gli stessi passi del metodo *miller_rabin*, dove al posto di restituire *falso* viene restituito -1 e con il controllo che x_j sia uguale a 1 e che il precedente, x_{j-1}, non sia congruo a -1 in modulo n: in questo caso, non si ha un fattore banale e si restituisce l'MCD.
Se da *miller_rabin_modificato* si ottiene un -1, allora l'algoritmo continua a generare un nuovo x nel medesimo intervallo, altrimenti è noto che n è composto e viene restituito il numero di iterazioni compiute dall'algoritmo.