Claudia Landi
Matricola: 7149573
Anno Accademico: 2023-2024

Sicurezza dei Dati e Privacy - Relazione secondo set di esercizi

1 Esercizi di programmazione

1.1 Esercizio 3.1

La soluzione dell'esercizio è stata implementata nel file *public_key_cryptography.py*, per la cui esecuzione è sufficiente eseguire il main.

a) L'algoritmo di Euclide esteso è stato implementato nel metodo *euclidean_extended* che prende come argomenti due numeri interi e restituisce il loro MCD e le relative identità di Bézout x e y, tali che $ax + by = MCD(a, b)$. Le variabili x, y, x_1 e y_1 rappresentano i coefficienti iniziali che saranno utilizzati per tenere traccia delle identità di Bézout durante l'iterazione. L'algoritmo continua ad iterare fintanto che il resto della divisione tra a e b non diventa zero e all'interno del ciclo esegue i passi dell'algoritmo di Euclide standard per calcolare il MCD, aggiornando anche i coefficienti. Alla fine dell'esecuzione della funzione, viene restituito il MCD e i valori x e y delle identità di Bézout.

b) Per l'algoritmo di esponenziazione modulare veloce è sato implementato il metodo *fast_exponentiation*, che prende in input tre numeri interi, a, e ed n e restituisce il risultato di $a^e mod(n)$ calcolato in modo efficiente, così da poter gestire anche numeri di grandi dimensioni. Per prima cosa l'esponente viene trasformato nel corrispondente in binario, dopodiché si itera con un ciclo attraverso le cifre di tale esponente a partire da quella più significativa. Per ogni iterazione, l'accumulatore di prodotti parziali d viene elevato al quadrato modulo n, inoltre, se il bit corrente risulta uguale a 1, d viene moltiplicato per la base in input, sempre modulo n.

Al termine delle iterazioni viene restituito il risultato contenuto nella variabile d.

c) Il test di Miller-Rabin utilizza un algoritmo di tipo Monte Carlo per determinare se un intero n è composto, in tal caso restituisce *vero*. Essendo un algoritmo probabilistico può commettere errori, infatti, mentre nel caso in cui restituisca *vero* si ha la certezza che n è composto, se restituisce *falso* non è certo che il numero sia effettivamente primo, con una probabilità di errore che non supera $\frac{1}{2}$ e che si può dimostrare essere al di sotto di $\frac{1}{4}$. Il test è implementato nel metodo *miller_rabin_test*, che prende in input due interi, uno è il numero di cui si vuole verificare la compostezza o primalità, n, l'altro, x, viene utilizzato durante il test. Per prima cosa l'algoritmo controlla se n è pari, in tal caso restituisce *vero*, in quanto un numero pari è sempre composto (ad eccezione di 2). In seguito tramite la funzione *find_odd_num* trova il numero dispari m e l'esponente r, tali che $2^r \cdot m = n$. Utilizza poi il risultato per determinare il termine iniziale x_0 utilizzando l'algoritmo di esponenziazione veloce per calcolare $x_0 = x^m mod(n)$. Se tale valore risulta uguale a ± 1, l'algoritmo termina restituendo *falso*, altrimenti entra in un ciclo di r iterazioni, in cui viene richiamato l'algoritmo di esponenziazione veloce per calcolare $x_i = x_{i-1}^2 mod(n)$. Per ogni valore di x_i viene controllato che $x_i \neq -1$, se la condizione viene violata l'algoritmo termina restituendo *falso*, se invece termina il ciclo senza che questo avvenga, viene restituito *vero*.

d) L'algoritmo per la generazione dei numeri primi è stato implementato nella funzione *generate_prime_number*, che prende in input un numero minimo e uno massimo come range per la ricerca del numero primo, e una soglia massima per la probabilità di errore. Infatti l'algoritmo di generazione dei numeri primi utilizza il test di Miller-Rabin, che essendo di tipo probabilistico può generare un errore, restituendo come primo un numero che in realtà è composto. Per diminuire la probabilità di errore è possibile ripetere più volte il test. Come prima cosa l'algoritmo genera casualmente un numero intero n, su cui applicherà il test di Miller-Rabin, e un numero x, che verrà utilizzato per testare la primalità. Dopodiché viene richiamato il test di Miller-Rabin più volte, finché si raggiunge la probabilità di errore desiderata. Se il test restituisce *falso* ad ogni iterazione, il numero primo viene restituito, altrimenti viene

scelto casualmente un nuovo numero su cui tentare il test, fino a che uno non risulta primo.

e) Per implemetare lo schema di cifratura e decifratura RSA, come prima cosa è stata definita una funzione *generate_keys*, che utilizza la funzione *generate_prime_factors* per generare i fattori primi p e q tramite l'algoritmo di generazione dei numeri primi descritto nel punto precedente. Dopodiché vengono generati il modulo n e gli esponenti pubblico e privato e e d. La funzione per generare le chiavi e gli esponenti prende in input un valore minimo e uno massimo, che corrispondono all'intervallo in cui si desidera che siano ricavati i fattori p e q. Per fare in modo di avere un modulo n realistico, tali limiti sono stati impostati rispettivamente a $10^{100} + 1$ e 10^{101}. Una volta trovate le chiavi, gli esponenti e il modulo, questi possono essere utilizzati per la cifratura e la decifratura. A questo punto, dopo aver generato casualmente un messggio, può essere richiamata la funzione per la cifratura *RSA_cipher*, passando come argomento il messaggio, l'esponente pubblico e il modulo, mentre per la decifratura può essere richiamata la stessa funzione, ma passando come esponente quello privato invece che quello pubblico. Si ha dunque:

$$c = E_e[m] = m^e mod(n)$$

$$m = E_d[c] = c^d mod(n)$$

La soluzione per la decifratura con l'ottimizzazione del teorema cinese del resto è stata invece implementata nella funzione *decrypt_RSA_with_CRT*, che prende in input il messaggio, i fattori p e q e i loro rispettivi inversi ($p^{-1}mod(q)$ e $q^{-1}mod(p)$). A questo punto vengono ricavati i due termini s_p ed s_q come segue:

$$S_p = m^{dmod(p-1)} mod(p)$$

$$S_q = m^{dmod(q-1)} mod(q)$$

il risultato viene poi ottenuto come:

$$S = [(q \cdot (q^{-1}mod(p)) \cdot S_p + (p \cdot (p^{-1}mod(q)) \cdot S_q)]mod(n)$$

1.2 Esercizio 3.2

a) Si vuole dimostrare che esiste l'indice j per cui $x_j = 1$ e $x_{j-1} \not\equiv_n -1$ e spiegare il motivo per cui quando l'algoritmo termina restituisce un fattore non banale di n. Per farlo si ricorda che per il teorema di Eulero $\forall x$ intero positivo, se $MCD(x, n) = 1$ si ha:

$$x^{\phi(n)} \equiv_n 1$$

Possiamo quindi affermare che tale indice j esiste, poiché si avrà per l'ultimo passo dell'algoritmo:

$$x_r \equiv_n x^{2^r \cdot m} \equiv_n x^{(e \cdot d - 1) mod(\phi(n))} = x^0 = 1$$

avendo elevato x^m al quadtrato per r volte, dove r e m sono stati ricavati tali che $e \cdot d - 1 = 2^r \cdot m$. Pertanto proseguendo con la sequenza dei quadrati di x^m, ad un certo punto si riscontrerà un x_j che elevato al quadrato sarà congruo a 1 modulo n. Se $x_j \neq -1$, allora $x_j - 1$ e $x_j + 1$ hanno ciascuno un fattore in comune con n distinto. Se n ha un fattorie in comune con $x_j - 1$ e $x_j + 1$ allora n divide il prodotto $(x_j - 1) \cdot (x_j + 1)$, ma non solo uno dei due fattori, altrimenti si avrebbe $x_j \equiv_n \pm 1$, che è un assurdo. I due fattori hanno quindi un fattore non banale in comune con n, che è possibile ricavare attraverso $MCD(x_j - 1, n)$ e $MCD(x_j + 1, n)$.

b) L'algoritmo è stato implementato nel file *decryption_ exponent_ attack.py*, per la cui esecuzione è sufficiente eseguire il main. L'implementazione è simile al test di Miller-Rabin, infatti come prima cosa viene richiamato il metodo *find_ odd_ num* già implementato nel file *public_ key_ cryptography.py*, per trovare un numero dispari m e l'esponente r, tali che $e \cdot d - 1 = 2^r \cdot m$. In seguito viene avviato un ciclo che si ripete finché non viene trovata una fattorizzazione per il modulo n. L'algoritmo procede selezionando casualmente un valore di x e controlla se il MCD tra x ed n è diverso da 1, in tal caso è stata trovata una fattorizzazione per n e vengono quindi restituiti i fattori p e q e il numero di iterazioni. Altrimenti si procede applicando il test di Miller-Rabin, implementato nel metodo *miller_ rabin_ algorithm*, che ad ogni passo controlla se $x_{j-1} \not\equiv_n -1$ e che $x_j = 1$. Se $x_{j-1} \not\equiv_n -1$ restituisce 0 e viene scelto un nuovo x in *decryptionexp*, se invece le condizioni precedenti sono soddisfatte, restituisce $MCD(x_{j-1} + 1, n)$ e

la fattorizzazione è stata trovata, quindi *decryptionexp* termina restituendo i fattori e il numero di iterazioni. Testando la funzione su 100 moduli RSA generati casualmente, si nota come le iterazioni restino effettivamente al di sotto di 2. I risultati di un'esecuzione sono riportati in tabella 1.

Numero medio iterazioni	Tempo medio (s)	Varianza tempi (s^2)	Deviazione standard (s)
1.53	0.0113	0.000 114 3	0.0107

Tabella 1: Statistiche delle esecuzioni

2 Esercizi di approfondimento

2.1 Esercizio 2.1

a) Si vuole dimostrare che, dati $x, y \in \mathbb{Z}_n^*$, con x testimone di Fermat di n e y non testimone di Fermat di n, xy mod n è un testimone di Fermat di n in $\mathbb{Z}_n^*$. Sappiamo per ipotesi che x è testimone di Fermat per n, ovvero $x^{n-1} \not\equiv_n 1$, mentre per y si ha $y^{n-1} \equiv_n 1$, essendo y un non testimone di Fermat per n. Per dimostrare che xy è un testimone di Fermat per n, si deve mostrare che $(xy)^{n-1} \not\equiv_n 1$. Per le osservazioni precedenti si ottiene:

$$(xy)^{n-1} \equiv_n x^{n-1}y^{n-1} \equiv_n z \cdot 1 \equiv_n z \neq 1$$

dove z è un qualsiasi intero diverso da 1. Pertanto il prodotto xy è un testimone di Fermat per n in $\mathbb{Z}_n^*$.

b) Si vuole dimostrare che se x è un testimone di Fermat e $x, y \in \mathbb{Z}_n^*$ sono due diversi non testimoni di Fermat, allora $xy \neq xy'$. Supponiamo per assurdo che $xy \equiv_n xy'$. Dato che x è un testimone di Fermat per n si ha che il loro MCD è pari a 1, pertanto esiste l'inverso moltiplicativo di x in $\mathbb{Z}_n^*$, quindi è possibile riscrivere la congruenza come segue:

$$y \equiv_n y'$$

ma questo è un assurdo, poiché se y e y' sono coprimi (da definizione perché in $\mathbb{Z}_n^*$) non possono essere congrui modulo n a meno che $y = y'$, che è falso per ipotesi. Pertanto si è dimostrato che $xy \neq xy'$.

c) Se sappiamo che n è non Carmichael, ovvero esiste almeno un testimone di Fermat x, possiamo concludere che in $\mathbb{Z}_n^*$ c'è almeno un testimone per ogni non testimone. Infatti per la proprietà dimostrata in a), $\forall y$ non testimone ho un testimone di Fermat dato da xy. Inoltre per la proprietà dimostrata in b) si ha che $xy \neq xy' \; \forall y \neq y'$, quindi i testimoni per y e y' sono due testimoni distinti tra loro. Si può quindi concludere che ci sono almeno $\frac{|Z_n^*|}{2} = \frac{\phi(n)}{2}$ testimoni di Fermat in $\mathbb{Z}_n^*$.

d) Si vuole ora dimostrare che se n è composto, dispari e non Carmichael la probabilità che l'algoritmo ritorni *vero* è almeno $\frac{1}{2}$. Se valgono tali ipotesi, come dimostrato in precedenza, si ha che in $\mathbb{Z}_n^*$ ci sono almeno $\frac{\phi(n)}{2}$ testimoni di Fermat, ovvero ho una probabilità di almeno $\frac{1}{2}$ di scegliere un testimone di Fermat. L'algoritmo sceglie un x nell'intervallo $(1, n-1)$ e controlla se ha fattori in comune con n. Se l'MCD risulta diverso da 1 l'algoritmo restituisce *vero*, in quanto n è sicuramente composto, altrimenti applica il test di Fermat, che restituisce *vero* se x è un testimone di Fermat. Dato che il numero di testimoni è almeno $\frac{1}{2}$, l'algoritmo restituirà *vero* con una probabilità pari o superiore a tale valore. Dato che nel test di Fermat e nell'algoritmo di Miller-Rabin un numero classificato come composto è certamente tale, la probabilità di errore è sicuramente inferiore a $\frac{1}{2}$.

2.2 Esercizio 2.4

Per la risoluzione dell'esercizio è stato utilizzato il codice Python riportato nel file *common_modulus_failure.py*. L'attacco "common modulus failure" consiste nel ricavare il messaggio originale m conoscendo unicamente due ciphertext c_1 e c_2, ottenuti cifrando m utilizzando due chiavi diverse e coprime tra loro e lo stesso modulo n. Tale attacco è possibile considerando che:

$$c_1^x \cdot c_2^y \equiv_n m^{xe_1} \cdot m^{ye_2} \equiv_n m^{xe_1+ye_2} = m$$

dove x e y sono i coefficienti dell'identità di Bézout, e_1 e e_2 sono i due esponenti pubblici coprimi tra loro. L'attacco è stato implementato nel metodo *common_modulus_failure_attack*, dove per prima cosa viene richiamato l'algoritmo di Euclide esteso (già implementato nel file *public_key_cryptography.py*) su e_1 e e_2, questo permette di ricavare i coefficienti di Bézout x e y e l'MCD tra i due esponenti. È stato quindi verificato che l'MCD fosse 1, in quanto se i due esponenti non sono coprimi tra loro non è possibile

portare a termine l'attacco, in questo caso viene sollevato un messaggio di errore. Se e_1 e e_2 risultano coprimi è possibile procedere, viene quindi fatto un controllo per verificare se si è ottenuto un coefficiente di Bézout negativo. Questo può risultare un problema nell'applicazione dell'algoritmo di esponenziazione veloce per ricavare $c_1^x mod n \cdot c_2^y mod n$, quindi è stato implementato il metodo *fix_negative_exp*, che trova il coefficiente negativo, ne calcola il valore assoluto e cacola l'inverso del corrispondente ciphertext, ovvero applica la proprietà:

$$x^{-y} = (x^{-1})^y$$

Infine è stato applicato l'algoritmo di esponenziazione veloce *fast_exponentiation*, già implementato nel file *public_key_cryptography.py* in modo da ricavare il messaggio in chiaro. Si ottiene dunque $m = 65535$.