Esercizi di programmazione

Set 2.

Di Giuseppe Martinelli

Matr: 7093926

Esercizio N 3.1 Implementazione di algoritmi per crittografia a chiave pubblica:

Svolgimento:

Il punto 1) viene soddisfatto dalla funzione *euclideEsteso(a,b)* (riga 23) la quale prende in input due interi e restituisce il loro massimo comun divisore, l'inverso di a modulo b e l'inverso di b modulo a. La funzione ricorsivamente calcola il GCD applicando ripetutamente il Teorema della Divisione e l'inverso di a e b. Da notare come il calcolo dell'inverso avvenga nella risalita delle chiamate ricorsive dove vi è allo stesso tempo anche una risalita della sequenza decrescente dei resti non negativi ottenuti proprio dalla applicazione ricorsive del Teorema della Divisione. Nello specifico, al passo i avremo che il valore della x (che rappresenta il valore del resto i-esimo) è ottenuto come:

$$x = y1 – (b // a) * x1$$

Questa espressione è equivalente alla seguente:

$$Ri = Ri-2 – Qi-1 * Ri-1$$

Infatti:

- Y1 = Corrisponde al resto della chiamata ricorsiva di due passi precedenti (poiché quando il metodo viene richiamato in y1 avremo il valore di a che è a sua volta il valore ottenuto due passi precedenti dal modulo b % a)
- B // A = È il valore del quoziente del passo precedente che viene calcolato
- X1 = E' il resto della chiamata ricorsiva precedente (cioè b % a)

Il punto 2) viene soddisfatto dalla funzione *esponenziazioneVeloce(base,esponente,modulo)* la quale prende in input una base, un esponente a cui elevare la base ed un modulo per cui dividere. La funzione in prima battuta espande l'esponente convertendolo nella sua espressione in base due. Ciò è necessario in quanto questa rappresentazione mi permette di poter effettuare computazioni più efficienti. Infatti, si eleva al quadrato la base e se ne prende il modulo. Poi si eleva questo numero ancora al quadrato e se ne prende il residuo modulo m.

Il punto 3) viene risolto dalla funzione *testMillerRabin(n)* la quale prendendo in input un numero ritorna un valore vero o falso a seconda del fatto che questo numero sia composto o meno. La funzione basa il suo funzionamento sulla funzione del punto 2) e sulla proposizione dei *resti quadratici*. La funzione inizialmente si calcola i valori m ed r per soddisfare l'equazione n – 1 = 2**r * m dove m ed r sono calcolati nel ciclo while m % 2 == 0. Questo ciclo che, finchè è vero, divide m per due e arrotonda per eccesso ed aumenta n

di uno. Questa operazione ci fa ottenere il numero dispari ed il resto che ci permettono di soddisfare l'equazione. Successivamente la funzione estrae un x compreso tra 1 e n – 1 e verifica che:

- X0 = x**m mod n sia diverso da 1 (altrimenti ritorna False)
- Per ogni r trovato in precedenza, verifica che xi**2 mod n sia diverso da -1 dove xi è il rimanente del modulo n applicato al passo precedente

Per il punto 4) è stata definita la funzione *generaNumeroPrimo()* la quale genera un numero casuale positivo grande 400 bit (121 cifre in base 10) ed attraverso operazioni di manipolazione di bit si assicura che sia dispari (ponendo il bit meno significativo uguale a 1) e sia nell'ordine di 2**k (ponendo il bit più significativo a 1). Successivamente la funzione verifica che questo numero sia effettivamente primo applicando k volte il test di Miller Rabin costruito al punto precedente.

Infine, il punto 5) è soddisfatto dalle righe 119 a 183 dove viene implementato RSA. Nel codice vengono prima generati due numeri primi con la funzione scritta nel punto 4), successivamente viene generato il modulo pubblico n = p * q. Essendo p e q primi, viene calcolato facilmente phi_n = (p – 1) * (q – 1). Il valore del fattore e = 3 è stato scelto arbitrariamente e da esso è stato calcolato il valore di d attraverso il metodo di euclide esteso la cui funzione è stata programmata precedentemente. Dopo questo punto, il codice genera un insieme di 200 messaggi casuali (cioè 200 interi casuali) che vengono criptati e decriptati utilizzando la funzione di esponenziazione veloce creata al punto 2). In questo frangente, il codice tiene traccia del tempo di decryption in quanto questo verrà confrontato con il tempo che invece il modulo RSA-CRT impiega per decriptare. Infatti, quest'ultimo sfruttando i risultati aritmetici ottenuti dal teorema cinese del resto permette di calcolare più velocemente i messaggi originali partendo dai ciphertext.

Esercizio N 3.3 Timing attack:

Svolgimento:

Il codice si prepone di trovare un esponente di decryption d segreto ricorrendo all'utilizzo di strumenti statistici per l'analisi del tempo di esecuzione dell'algoritmo di esponenziazione. Parte del codice utilizza funzioni del modulo *TimingAttack.py* fornito dal docente che simula il tempo di esecuzione di un dispositivo sotto questa tipologia di attacco (quindi simula il tempo di esponenziazione con un d segreto). Inoltre, poiché l'attaccante non conosce il ciphertext che la vittima sta utilizzando per la decryption allora non resta altro che generare c ciphertexts casuali e procedere per tentativi.

Il codice istanzia un oggetto della classe TimingAttack la quale mette a disposizione i metodi *victimdevice* e *attackerdevice* che verranno utilizzati successivamente dalla funzione *attack(ta,N,d_len)* la quale prende in input:

- **Ta** che rappresenta una instanza della classe TimingAttacck
- **N** Il numero di estrazione i.i.d che si operano per calcolare la varianza
- **D_len** che rappresenta la lunghezza della chiave d che è nota

All'interno di questa funzione vi è un ciclo che simula l'estrazione di N variabili indipendenti per cui si effettua inferenza statistica, in particolare si va a calcolare dato un ciphertext c (generato casualmente) la differenza dei tempi di esecuzione nel caso in cui si ipotizzi che il bit della variabile d sia uguale a 1 oppure

uguale a 0. Il calcolo della varianza viene effettuato nella funzione *varianza(N,t)* che prende in input i numeri di tentativi passati alla funzione attack ed i tempi di esecuzione della funzione di esponenziazione utilizzando il d' con un particolare bit. Alla fine del calcolo delle varianze il codice decide quale è il bit migliore che più si avvicina al valore contenuto nel valore d che stiamo cercando. In altre parole il codice si costruisce un vettore d' basandosi sui tempi di esecuzione che più si avvicinano ai tempo di esecuzione del d non conosciuto.

Infine, la funzione ritorna d' e si verifica la correttezza dell'esponente trovato attraverso una funzione di test del modulo TimingAttack.py chiamata *test(d)* che ritorna un messaggio in console in base alla correttezza dell'esponente