Programmazione Data Security 2

Jacopo Manetti

May 2023

1 Implementazione di algoritmi per crittografia a chiave pubblica

1.1 Traccia

Programmare in Python i seguenti algoritmi discussi in classe.

1. Algoritmo di Euclide esteso.
2. Algoritmo di esponenziazione modulare veloce.
3. Test di Miller-Rabin.
4. Algoritmo per la generazione di numeri primi.
5. Schema RSA, con e senza ottimizzazione CRT.

Gli algoritmi devono essere in grado di manipolare numeri di dimensione realistica $\left(> 10^{100}\right)$. Non è permesso implementare gli algoritmi in questione richiamando semplicemente le corrispondenti funzioni predefinite. E' invece lecito, per esempio, usare package per la gestione di numeri molto grandi, e per le operazioni,,$+ - \times$ in aritmetica modulo n. Servendosi della funzione 5, una volta fissato un modulo RSA realistico, testare la funzione di decryption su 100 ciphertext scelti casualmente. Confrontare le prestazioni, in termini di tempo di esecuzione, delle versione senza e con CRT (NB: i valori che si usano nella formula CRT vanno precomputati).

1.2 Svolgimento

Questo codice contiene diverse funzioni utilizzate per la crittografia, in particolare per l'algoritmo di crittografia RSA. Qui di seguito è presente una descrizione dettagliata di ogni funzione.

1. **algoritmo_euclide_esteso(a, b):** Questa funzione implementa l'algoritmo di Euclide esteso per trovare il massimo comune divisore (MCD) di due numeri e i coefficienti di Bézout. I coefficienti di Bézout sono due interi x e y tali che $ax + by = MCD(a, b)$.

2. **esponenziazione_modulare_veloce(base, exp, mod):** Questa funzione implementa l'esponenziazione modulare, che è un'operazione chiave in molti algoritmi crittografici, incluso RSA. Invece di calcolare prima l'elevamento a potenza (**base** elevato a **exp**) e poi applicare il modulo **mod**, questa funzione suddivide l'operazione in sotto-problemi, calcolando l'elevamento a potenza e l'applicazione del modulo su potenze più piccole fino a raggiungere il valore di partenza. Questo riduce sia il tempo di elaborazione che la quantità di memoria richiesta, poiché non è necessario memorizzare grandi numeri intermedi. Il risultato di questa operazione viene restituito come output.

3. **miller_rabin(n, k)**: Questa funzione implementa il test di primalità di Miller-Rabin, che è un algoritmo probabilistico utilizzato per determinare se un numero n è primo. L'algoritmo prende in input un numero n da testare per la primalità e un numero k che rappresenta il numero di round di test da eseguire. Miller-Rabin opera scomponendo $n-1$ come $2^r * d$ con d dispari. Poi, per k round, sceglie un numero casuale a tra 2 e $n-2$, calcola $a^d mod n$ e verifica una serie di condizioni per determinare se n è composto. Se in qualsiasi round l'algoritmo determina che n è composto, la funzione ritorna False. Se dopo k round, l'algoritmo non ha ancora determinato che n è composto, allora ritorna True, indicando che n è probabilmente primo, di solito un k pari a 5 o 10 è sufficiente.

4. **genera_primo(k):** Questa funzione genera un numero primo casuale di k cifre binarie. Lo fa generando numeri casuali di k cifre binarie fino a quando non ne trova uno che passa il test di Miller-Rabin. Restituisce il primo numero primo trovato.

5. Lo schema RSA è composto da 3 funzioni di seguito riportate:

   rsa_keygen(p, q): Questa funzione genera una coppia di chiavi RSA (una chiave pubblica e una chiave privata) dati due numeri primi **p** e **q**. La chiave pubblica è una coppia (e, n) e la chiave privata è una coppia (d, n), dove **n** è il prodotto di **p** e **q**, **e** è un intero coprimo con (p-1)*(q-1), e **d** è l'inverso moltiplicativo di **e** modulo (p-1)*(q-1)

   rsa_encrypt(plain_text, public_key): Questa funzione cifra un messaggio in chiaro utilizzando una chiave pubblica RSA.

   rsa_decrypt(cipher_text, private_key): Questa funzione decifra un messaggio cifrato utilizzando una chiave privata RSA.

6. Lo schema RSA che sfrutta l'ottimizzazione tramite CRT (Teorema cinese del resto) è composto dalle seguenti 3 funzioni:

   rsa_keygen_crt(p, q): Questa funzione è simile a rsa_keygen(p, q), ma restituisce una chiave privata in una forma estesa che include i valori aggiuntivi necessari per l'ottimizzazione del Teorema Cinese del Resto (CRT).

rsa_encrypt_crt(plain_text, public_key): Questa funzione cifra un messaggio in chiaro utilizzando una chiave pubblica RSA. Questa funzione è identica alla funzione rsa_encrypt.

rsa_decrypt_crt(cipher_text, private_key): Questa funzione decripta un messaggio cifrato **cipher_text** utilizzando la chiave privata **private_key** attraverso l'algoritmo RSA, ma con l'ottimizzazione CRT. L'ottimizzazione CRT sfrutta la proprietà che i due numeri primi utilizzati nel processo di generazione della chiave RSA sono unici e sconosciuti a chiunque tranne il proprietario della chiave. Quindi, piuttosto che calcolare un esponente di decriptazione su un grande modulo n (il prodotto dei due numeri primi), l'ottimizzazione CRT consente di dividere il calcolo in due parti, ciascuna su un modulo più piccolo (ovvero, i numeri primi stessi),il che velocizza notevolmente l'operazione. La funzione restituisce il testo in chiaro.

Infine, il codice presenta un esempio d'uso che chiede all'utente di selezionare quale funzione eseguire e fornisce le istruzioni su come inserire i dati necessari. Inoltre viene fornita una settima opzione per confrontare le prestazioni, in termini di tempo di esecuzione, delle versione senza e con CRT su 100 ciphertext scelti casualmente.

2 Implementazione del Timing attack

2.1 Traccia

Il modulo Python TimingAttackModule.py, disponibile sulla pagina Moodle del corso, simula le seguenti funzionalità, relative ad un dispositivo sotto attacco (la vittima), che implementa la decryption RSA, cioè la funzione $c \mapsto c^d \bmod n$, per un certo esponente segreto d e un modulo n fissati e incorporati nel dispositivo simulato, e ad un attaccante che ha preso di mira quel dispositivo.

- ta=TimingAttack(): crea un oggetto ta di classe TimingAttack. Tale oggetto simula sia le funzionalit del dispositivo sotto attacco, con la sua chiave privata d, che il dispositivo dell'attaccante, come descritto di seguito.
- ta.victimdevice(c): preso un intero c, restituisce il tempo di esecuzione in μs dell'algoritmo di esponenziazione modulare relativo al calcolo di $c^d \bmod n$. 14
- ta.attackerdevice(c,v): preso un intero c e una lista v di $k \geq 1$ bit, restituisce il tempo di esecuzione dell'algoritmo di esponenziazione modulare relativo al calcolo di $c^{d'} \bmod n$, dove d' è il numero intero rappresentato da v in base 2. Dunque in questo caso l'algoritmo esegue esattamente k iterazioni del ciclo for.

Servendosi di queste due funzioni, programmare in Python un algoritmo di Timing Attack come visto a lezione che, preso in input l'oggetto ta, recuperi

l'esponente segreto d incorporato nel dispositivo della vittima. Si tenga presente quanto segue.

1. L'esponente segreto d è di 64 bit, di cui quello più a sinistra è 1 . Nell'algoritmo, gli esponenti sono rappresentati come liste di bit, scandite da sinistra (posizione 0) a destra.

2. Un esponente candidato d', rappresentato come lista di bit, può essere testato invocando la funzione ta.test(d'), messa a disposizione anch'essa dal modulo. Essa dice se la frazione di bit corretti di d' rispetto all'esponente segreto d è inferiore al 75%, oppure superiore al 75% ma inferiore al 100%, oppure del 100% (cioè se $d = d'$)

3. I tempi di esecuzioni delle operazioni di moltiplicazione all'interno dell'algoritmo di esponenziazione non sono fissi, ma variano al variare di c. Se c è scelto casualmente, tali tempi seguono una distribuzione Gaussiana di media 1000 e deviazione standard 50.

Le funzionalità del modulo possono essere importate dalla console Python tramite il comando from TimingAttackModule import * (NB: assicurarsi che il file TimingAttackModule.py si trovi nella directory di lavoro di Python. E' possibile modificare la directory di lavoro tramite il comando os . chdir (path) del package os).

2.2 Svolgimento

Il codice Python per l' implementazione del timing attack si basa su due funzioni principali: **compute_variance_from_observations** e **perform_timing_attack**.

compute_variance_from_observations(exp, ta): Questa funzione prende due argomenti come input:
exp: Una lista di numeri binari che rappresenta l'esponente corrente nell'attacco.
ta: Un'istanza dell'oggetto $TimingAttack$, che ha due metodi importanti: $victimdevice(ct)$ e $attackerdevice(ct, exp)$. Questa funzione genera 3000 ciphertext casuali a 64 bit e misura il tempo di esecuzione del *victimdevice* per ciascuno di questi numeri. Quindi, per ogni numero, sottrae il tempo impiegato dall'*attackerdevice* per eseguire la stessa operazione per i due possibili valori dell'esponente (0 e 1). Memorizza queste differenze di tempo in due liste separate (una per quando l'esponente è 0 e una per quando l'esponente è 1). Infine, calcola e restituisce le varianze di queste due liste di differenze di tempo.

perform_timing_attack(): Questa funzione inizia inizializzando un'istanza dell'oggetto $TimingAttack$ e una lista *exp* con un singolo elemento, 1. Quindi, esegue un ciclo per 64 volte (lunghezza dell'esponente). In ogni iterazione, chiama $compute\_variance\_from\_observations$ per calcolare la varianza delle differenze di tempo per un esponente di 0 e 1. Se la varianza per l'esponente 0 è minore, aggiunge 0 all'esponente corrente *exp*, altrimenti aggiunge 1. Dopo

ciascuna iterazione, stampa l'esponente corrente. Alla fine delle 64 iterazioni, stampa l'esponente finale recuperato e chiama il metodo test dell'oggetto $TimingAttack$ passandogli l'esponente recuperato.