Corso di *Data Security & Privacy*[*]
Esercizi su crittografia a chiave pubblica

Michele Boreale
Università di Firenze
Dipartimento di Statistica, Informatica, Applicazioni
michele.boreale@unifi.it

1 Esercizi di verifica

I seguenti esercizi vertono sugli argomenti esposti nei capitoli 7 a 11 delle *Note*. Di diversi esercizi viene fornita una traccia di svolgimento.

1.1 Aritmetica modulare, MCD, teorema di Eulero

1. Si dimostri che vale la seguente relazione, per a, b interi non entrambi nulli:

$$\text{mcm}(a,b) = \frac{|a \times b|}{\text{MCD}(a,b)}.$$

   Traccia. Si ricordi che $\text{mcm}(a,b) = \Pi_p p^{\max\{a_p, b_p\}}$ (dove con a_p, b_p indichiamo qui gli esponenti dei fattori primi nella fattorizzazione di $|a|$ e $|b|$). Allora

$$\frac{|a \times b|}{\text{MCD}(a,b)} = \frac{\Pi_p p^{a_p + b_p}}{\Pi_p p^{\min\{a_p, b_p\}}} = \Pi_p p^{a_p + b_p - \min\{a_p, b_p\}} = \dots$$

2. Si trovi il MCD delle seguenti coppie di numeri e, dove possibile, l'inverso moltiplicativo dell'uno rispetto all'altro, usando l'algoritmo di Euclide esteso: $(124, 32)$, $(575, 34)$, $(1028, 56)$.

3. Provare che, per ogni $n \geq 1$, $\text{MCD}(n, n+1) = 1$.

   Traccia. Si consideri un qualsiasi divisore comune d di n e $n+1$. Dal momento che d divide n e $n+1$, divide anche la loro differenza.

4. Siano a e b interi non entrambi nulli. Provare che $d = \text{MCD}(a,b)$ è il più piccolo intero positivo che può essere espresso come $ax + by$, per qualche $x, y \in \mathbb{Z}$.

   Traccia. Dall'identità di Bezout, sappiamo che d può essere espresso in quella forma. Si consideri un altro c che può essere espresso in quella forma. Allora $d|c$ (perché?), e dunque...

5. Siano a e b due interi, non entrambi nulli, dei quali si sa che $15 = ax + by$ per qualche $x, y \in \mathbb{Z}$. Dire quali sono i possibili valori di $d = \text{MCD}(a,b)$.

[*]Laurea Magistrale in Informatica, Università di Firenze, A.A. 2022-2023.

6. Dire se la seguente affermazione è vera o falsa, mostrando nel caso un controesempio: se c divide $a \times b$, allora c divide a, oppure c divide b.

7. Dimostrare che se d divide $a \times b$ e $\mathrm{MCD}(d, a) = 1$, allora d divide b.

   Traccia. Sappiamo che per ogni primo p: $d_p \leq a_p + b_p$ (d divide $a \times b$) e che $\min\{d_p, a_p\} = 0$ ($\mathrm{MCD}(d, a) = 1$), dunque in particolare, se $d_p > 0$ allora $a_p = 0$. Da questo segue che, per ogni p, $d_p \leq b_p$ (perché?). Da cui la tesi.

8. Siano a, b numeri interi positivi. Si dimostri che $a \equiv_9 b$ se e solo se $C(a) \equiv_9 C(b)$, dove $C(x)$ denota la somma delle cifre di x in base 10 (es. $C(386) = 17$).

9. Si dimostri che un numero intero di k cifre in base 10, $a = (d_{k-1}, ..., d_1, d_0)_{10}$, è divisibile per 11 se e solo se $\sum_{i \text{ dispari}} d_i \equiv_{11} \sum_{j \text{ pari}} d_j$.

10. Si dimostri che, nel Teorema di Eulero, l'ipotesi che $\mathrm{MCD}(a, n) = 1$ è necessaria. Ovvero, si trovino a e n con $(a, n) > 1$ e tali che $a^{\phi(n)} \not\equiv_n 1$. **Traccia.** Sia $p > 1$ un divisore comune di a ed n. Per qualsiasi intero $k > 0$, p divide a^k; se dunque fosse $a^k \equiv_n 1$...

11. Usando il Teorema di Eulero, dimostrare che, per potenze con basi relativamente prime al modulo n, l'esponente può essere ridotto modulo $\phi(n)$, ovvero: per $(a, n) = 1$ e $x > 0$, vale che $a^x \equiv_n a^{x \bmod \phi(n)}$.

    Traccia. Per il teorema della divisione $x = j \cdot \phi(n) + r$ per qualche j e $r = x \bmod \phi(n)$. Dunque, $a^x = a^{j \cdot \phi(n) + r} = (a^{\phi(n)})^j \cdot a^r = \cdots$.

12. Usando il fatto che se $\mathrm{MCD}(a, b) = 1$ allora $\phi(ab) = \phi(a)\phi(b)$ e che $\phi(p^i) = p^i - p^{i-1}$ (per p primo e $i > 0$), dare una formula per $\phi(n)$, basata sulla fattorizzazione di n.

13. Usando la formula dell'esercizio precedente, provare che per ogni $n > 2$, $\phi(n)$ è pari.

14. Siano a, n interi con $n \geq 1$ e $(a, n) = 1$. Per un intero *qualsiasi* $x \in \mathbb{Z}$ (quindi eventualmente negativo), possiamo definire

$$a^x \stackrel{\text{def}}{=} a^{x \bmod \phi(n)}$$

    dove notiamo che a destra dell'uguaglianza, anche quando $x < 0$, abbiamo un esponente $x \bmod \phi(n) \geq 0$. Dopo aver controllato che questa definizione sia consistente con la normale definizione quando $x \geq 0$, provare che essa obbedisce alle usuali leggi delle potenze, cioè che per ogni $x, y \in \mathbb{Z}$:

$$\begin{aligned} a^{x+y} &\equiv_n a^x a^y \\ (a^x)^y &\equiv_n a^{xy}\,. \end{aligned}$$

    Traccia. Si tratta di applicare il teorema di Eulero.

15. Siano a e $n \geq 1$ interi. Si dimostri che $a^{-1} \equiv_n a^{\phi(n)-1}$, quando $a^{-1} \bmod n$ esiste. Verificare questo fatto calcolando $1088^{-1} \bmod 9$.

1.2 Esponenziazione modulare

1. Applicare l'algoritmo di esponenziazione veloce al calcolo dei seguenti elevamenti a potenza: $6^{124} \bmod 10$, $9^{345} \bmod 11$, $9^{1025} \bmod 10$. Illustrare i vari passaggi, mediante una tabella che riporti per ogni riga il valore delle variabili d e c alla fine di ogni iterazione.

2. Nella versione vista in classe dell'algoritmo di esponenziazione per il calcolo di $a^d \bmod n$, i bit dell'esponente vengono scanditi da sinistra a destra, ovvero dal più significativo al meno significativo. Descrivere una versione dell'algoritmo dove i bit dell'esponente vengono scanditi da destra a sinistra.

   Traccia. Consideriamo il seguente algoritmo (in sintassi Python).

```
w = 1
z = a
for j = 0 to k-1:
        if d[j] == 1:
             w= w*z mod n
        z= z*z mod n
return w
```

   Si noti prima di tutto che, all'inizio dell'iterazione j-ma (per $j = 0, ..., k-1$), la variabile z contiene il valore $z_j \stackrel{\text{def}}{=} a^{2^j} \bmod n$. Chiaramente $z_j^{d_j}$ vale 1 se $d_j = 0$, mentre vale $z_j = a^{2^j} \bmod n$ se $d_j = 1$. Per costruzione, il valore finale della variabile w è allora dato da

$$w_{fin} \stackrel{\text{def}}{=} z_0^{d_0} \times z_1^{d_1} \times \cdots \times z_{k-1}^{d_{k-1}} .$$

   Consideriamo ora l'espansione binaria dell'esponente, $d = \sum_{j=0}^{k-1} d_j \times 2^j$. Abbiamo

$$\begin{aligned}
a^d &= a^{\sum_{j=0}^{k-1} d_j \times 2^j} \\
&= (a^{2^0})^{d_0} \times (a^{2^1})^{d_1} \times \cdots \times (a^{2^{k-1}})^{d_{k-1}} \\
&\equiv_n \left((a^{2^0})^{d_0} \bmod n\right) \times \left((a^{2^1})^{d_1} \bmod n\right) \times \cdots \times \left((a^{2^{k-1}})^{d_{k-1}} \bmod n\right) \\
&\equiv_n \left((a^{2^0} \bmod n)^{d_0}\right) \times \left((a^{2^1} \bmod n)^{d_1}\right) \times \cdots \times \left((a^{2^{k-1}} \bmod n)^{d_{k-1}}\right) \\
&\equiv_n z_0^{d_0} \times z_1^{d_1} \times \cdots \times z_{k-1}^{d_{k-1}} \\
&\equiv_n w_{fin} .
\end{aligned}$$

1.3 Teorema Cinese del Resto

1. Usando il Teorema Cinese del Resto (Chinese Remainder Theorem, CRT), dimostrare che se $(m, n) = 1$ allora $\phi(m \cdot n) = \phi(m) \cdot \phi(n)$.

   Traccia. Assumendo senza perdita di generalità che $m, n > 1$, è sufficiente dimostrare che $\phi(m \cdot n) = |\mathbb{Z}^*_{m \cdot n}| = |\mathbb{Z}^*_m| \cdot |\mathbb{Z}^*_n| = \phi(m) \cdot \phi(n)$. Per far questo, mostriamo che la funzione $f : \mathbb{Z}^*_{m \cdot n} \to \mathbb{Z}^*_m \times \mathbb{Z}^*_n$ definita come

$$f(x) \stackrel{\text{def}}{=} \langle x \bmod m, x \bmod n \rangle \qquad (x \in \mathbb{Z}^*_{m \cdot n})$$

   è biettiva. Intanto bisogna provare che la funzione è ben definita, cioè che effettivamente $x \bmod m \in \mathbb{Z}^*_m$ e $x \bmod n \in \mathbb{Z}^*_n$: ma questo segue dal fatto che $(x, m \cdot n) = 1$ e

dall'algoritmo di Euclide. La funzione è poi iniettiva: infatti, dati $x, x' \in \mathbb{Z}^*_{m\cdot n}$ tali che $f(x) = f(x') = \langle a, b\rangle$, si vede che sia x che x' sono soluzioni del sistema

$$\begin{cases} X \equiv_m a \\ X \equiv_n b \end{cases}$$

e dunque, per il CRT-parte unicità, $x \equiv_{m\cdot n} x'$, e dunque $x = x'$. D'altra parte, la funzione è anche suriettiva. Infatti, presa una qualsiasi coppia $\langle a, b\rangle \in \mathbb{Z}^*_m \times \mathbb{Z}^*_n$, il sistema di cui sopra ha una soluzione $x \in \mathbb{Z}_{m\cdot n}$, per il CRT-parte esistenza. Inoltre, per Euclide, si verifica che $(x, m) = (m, x \bmod m) = (m, a) = 1$ e analogamente $(x, n) = 1$, e dunque $(x, m \cdot n) = 1$: perciò in effetti $x \in \mathbb{Z}^*_{m\cdot n}$. Per definizione di soluzione del sistema e della funzione f, si ha infine $f(x) = \langle a, b\rangle$.

2. Il metodo del CRT *speed up* visto in classe può essere usato per velocizzare il calcolo della firma digitale $S = m^d \bmod n$ (oppure della decryption $m = c^d \bmod n$), con $n = p\cdot q$ e p, q primi distinti. Applicare il metodo nei seguenti casi: $5^7 \bmod 55$, $9^{23} \bmod 143$.

   Traccia. Ricordiamo che la tecnica consiste nel calcolare separatamente $S_p \stackrel{\text{def}}{=} m^d \bmod p$ e $S_q \stackrel{\text{def}}{=} m^d \bmod q$, per poi ricombinare questi due valori usando il CRT. Infatti, poiché per definizione $m^d = j \cdot p \cdot q + S$, e dunque $S = m^d - j \cdot p \cdot q$, allora S è (l'unica) soluzione in $\mathbb{Z}_n$ del sistema

$$\begin{cases} X \equiv_p S_p \\ X \equiv_q S_q\,. \end{cases}$$

   Dunque, dalla formula vista nella prova del CRT, deve valere

$$S = \mathrm{CRT}(S_p, S_q) \stackrel{\text{def}}{=} \left\{q \cdot (q^{-1} \bmod p) \cdot S_p \;+\; p \cdot (p^{-1} \bmod q) \cdot S_q \right\} \bmod n\,.$$

   Il vantaggio di questo metodo è il seguente. Posto $k = \log_2(n)$ (il numero di bit del modulo), il costo dell'algoritmo di esponenziazione veloce è limitato superiormente da ck^3, per una certa costante c. Essendo p e q di $\approx k/2$ bit ciascuno, con il CRT speed up, il costo è dunque pari a quello di due esponenziazioni modulari (trascurando il costo delle moltiplicazioni e della riduzione modulare finali; si noti che $p, q, p^{-1} \bmod q, q^{-1} \bmod p$ possono essere pre-calcolati), condotte però su moduli di $k/2$ bit ciascuno, ovvero:

$$\approx c\left(\frac{k}{2}\right)^3 + c\left(\frac{k}{2}\right)^3 \;=\; \frac{c}{4}k^3\,.$$

3. Si dimostri che se n è un prodotto di primi distinti, $n = p_1 \cdots p_k$, e per ogni $i = 1, ..., k$ vale che $(p_i - 1)|(n - 1)$, allora n è Carmichael.

   Traccia. Bisogna provare che per ogni x con $(x, n) = 1$, vale che $x^{n-1} \equiv_n 1$. Questa uguaglianza può essere 'proiettata' sui primi p_i tramite il CRT. Poi si applica Fermat.

1.4 Test di primalità

1. Un certo algoritmo probabilistico consiste di un ciclo in cui, ad ogni singola iterazione, la probabilità di terminare vale $\lambda < 1$. Inoltre le varie iterazioni del ciclo sono probabilisticamente indipendenti.

   (a) Si provino le seguenti formule per un generico μ con $|\mu| < 1$: $\sum_{i=0}^{\infty} \mu^i = \frac{1}{1-\mu}$ e $\sum_{i=1}^{\infty} i \cdot \mu^{i-1} = \frac{1}{(1-\mu)^2}$.

(b) Sfruttando il punto precedente, provi che il numero medio di iterazioni dell'algoritmo è pari a $1/\lambda$.

Traccia. Per il punto (b), si tenga presente quanto segue: la probabilità che l'algoritmo termini esattamente all'iterazione i-ma ($i \geq 1$) è $(1-\lambda)^{i-1}\lambda$ (perché?).

2. Si consideri l'algoritmo visto in classe per generare un numero primo $n > 2$ di k bit, cioè con $2^{k-1} < n < 2^k$. Esso può essere concisamente descritto come segue.

   1. Scegli a caso un numero n dispari con $2^{k-1} < n < 2^k$.
   2. n è primo? Se sì, termina restituendo n; altrimenti torna al passo 1.

   Per il passo 2, si supponga di disporre di un test di primalità deterministico.

   (a) Stimare il numero N di primi compresi tra 2^{k-1} e 2^k. Si tenga presente la stima per

   $$\pi(n) \stackrel{\text{def}}{=} |\{x : 0 < x \leq n \text{ e } n \text{ è primo }\}|$$

   data da

   $$\pi(n) \sim \frac{n}{\ln n}\,.$$

   (b) Stimare la probabilità λ che l'algoritmo termini in una singola iterazione.

   (c) Sfruttando i risultati dell'esercizio precedente, si stimi il numero medio di iterazioni dell'algoritmo.

   (d) Stimare il numero medio di iterazioni per produrre un primo di 4096 bit.

   Traccia. Per il punto (a), il numero N di primi compresi tra 2^{k-1} e 2^k può essere così stimato:

   $$\begin{aligned} N &= \pi(2^k) - \pi(2^{k-1}) \\ &= \pi(2 \cdot 2^{k-1}) - \pi(2^{k-1}) \\ &\approx \frac{2 \cdot 2^{k-1}}{\ln 2 + (k-1)\ln 2} - \frac{2^{k-1}}{(k-1)\ln 2} \\ &\approx \frac{2 \cdot 2^{(k-1)}}{(k-1)\ln 2} - \frac{2^{k-1}}{(k-1)\ln 2} \\ &= \frac{2^{k-1}}{(k-1)\ln 2}\,. \end{aligned}$$

   Per quanto riguarda (b), considerando che i dispari di k bit sono in numero di $\frac{1}{2}(2^k - 2^{k-1}) = 2^{k-2}$, la probabilità che l'intero n scelto al passo 1 sia primo è dunque

   $$\lambda = \frac{N}{2^{k-2}} \approx \frac{2}{(k-1)\ln 2}\,.$$

3. Applicare il test di Rabin ai seguenti interi: 249, 787, 1105. In ciascun caso, dire di che forma è la sequenza ottenuta, e, se possibile, scomporre il numero dato in due fattori non banali.

4. Sia $A(\cdot)$ un test Monte-Carlo di compositness. Si supponga che la probabilità che l'algoritmo dia la risposta scorretta, dato che l'input $n > 1$ è un intero composto dispari di k bit, sia $\leq \epsilon < 1$. Cioè:

$$\Pr\left(A(n) = 'NO' \,|\, n \text{ è un intero composto dispari di } k \text{ bit }\right) \leq \epsilon\,.$$

   (a) Si stimi la probabilità di arrivare ad una conclusione errata, dato che l'algoritmo ha risposto 'NO' in m esecuzioni indipendenti in successione con lo stesso input n.

   (b) Nel caso del test di Rabin, dove $\epsilon \leq \frac{1}{4}$, confrontare la probabilità del punto (a) con ϵ^m, per $m = 2, 5, 10, 20$, quando $k = 512$ bit.

   Traccia. Per il punto (a), si denoti con a l'evento

$$"n \text{ è un intero composto dispari di } k \text{ bit}"$$

   e con $b^{(m)}$ l'evento

$$"A(n) \text{ risponde 'NO' } m \text{ volte in successione}".$$

   La probabilità cercata è dunque $\Pr(a|b^{(m)})$. Per ipotesi $\Pr(b^{(1)}|a) \leq \epsilon$, da cui, per l'indipendenza delle esecuzioni, si ha che $\Pr(b^{(m)}|a) \leq \epsilon^m$. Si procede ora usando il Teorema di Bayes. La stima di $\Pr(a) \approx 1 - \frac{2}{(k-1)\ln 2}$ può essere desunta da un esercizio precedente. La probabilità $\Pr(b^{(m)})$ può essere scomposta come:

$$\Pr(b^{(m)}) = \Pr(b^{(m)}|a)\Pr(a) + \Pr(b^{(m)}|\overline{a})\Pr(\overline{a})$$

   dove $\overline{a}$ è l'evento complementare di a, ovvero, "n è primo". Si noti anche che $\Pr(b^{(m)}|\overline{a}) = 1$ (perché?). Da queste considerazioni si arriva a:

$$\begin{aligned}
\Pr(a|b^{(m)}) &= \frac{\Pr(b^{(m)}|a)\Pr(a)}{\Pr(b^{(m)}|a)\Pr(a) + \Pr(\overline{a})} \\
&\approx \frac{\Pr(b^{(m)}|a)(1 - \frac{2}{(k-1)\ln 2})}{\Pr(b^{(m)}|a)(1 - \frac{2}{(k-1)\ln 2}) + \frac{2}{(k-1)\ln 2}} \\
&= \frac{\Pr(b^{(m)}|a)((k-1)\ln 2 - 2)}{\Pr(b^{(m)}|a)((k-1)\ln 2 - 2) + 2} \\
&= \frac{\Pr(b^{(m)}|a)((k-1)\ln 2 - 2)}{\Pr(b^{(m)}|a)((k-1)\ln 2 - 2 + \frac{2}{\Pr(b^{(m)}|a)})} \\
&= \frac{(k-1)\ln 2 - 2}{(k-1)\ln 2 - 2 + \frac{2}{\Pr(b^{(m)}|a)}} \\
&\leq \frac{(k-1)\ln 2 - 2}{(k-1)\ln 2 - 2 + 2\epsilon^{-m}}\,.
\end{aligned}$$

   Per il punto (b), sostituendo a ϵ il valore $\frac{1}{4}$ nell'ultima espressione, otteniamo la limitazione

$$\Pr(a|b^{(m)}) \leq \frac{(k-1)\ln 2 - 2}{(k-1)\ln 2 - 2 + 2^{2m+1}}\,.$$

1.5 RSA ed El Gamal

1. Generare una coppia di chiavi pubblica-privata (K^+, K^-) per ciascuno dei seguenti moduli RSA: $n = 13 \times 17$, $n = 19 \times 23$, $n = 37 \times 41$. In ciascun caso, calcolare quindi l'encryption $c = E_{K^+}[m]$ per il messaggio $m = 15$. Verificare poi che effettivamente $D_{K^-}[c] = m$.

2. L'utente Bob genera un modulo RSA $n = pq$ e due esponenti pubblici, e_1 ed e_2 relativi a questo modulo. Egli adotta quindi un sistema di doppia cifratura, per cui, se Alice vuole inviargli un messaggio cifrato, dovra calcolare e inviargli il ciphertext come $c_2 = c_1^{e_2} \bmod n$, con $c_1 = m^{e_1} \bmod n$. Dire se questo sistema di doppia cifratura è più sicuro rispetto ad una ordinaria cifratura RSA.

   Traccia. Discutere se questo schema a chiave doppia è equivalente ad uno schema a chiave singola, per lo stesso modulo.

3. Alice ha inviato a Bob un messaggio cifrato contenente un importante segreto, $c = m^e \bmod n$, dove $K_B^+ = \langle e, n \rangle$. Successivamente, Bob, catturato dal nemico e torturato, si dichiara disposto a decifrare qualsiasi ciphertext c' gli verrà chiesto di decifrare, e di riportare la risposta al richiedente, *a patto che* $c' \neq c$. Dire come il nemico può recuperare facilmente m, senza torturare Bob ulteriormente.

   Traccia. Basta 'mascherare' c, moltiplicandolo per un opportuno fattore prima di inviarlo a Bob. Cf. anche la tecnica del *blinding* contro i timing attack vista a lezione.

4. **(Fault attack su rsa con CRT (Boneh, Lipton, DeMillo 1996))**. Nel calcolo della firma digitale $S = m^d \bmod n$ (o della decryption $m = c^d \bmod n$) con la tecnica del CRT speed-up, i valori S_p e S_q vengono calcolati separatamente e poi ricombinati ponendo $S = \mathrm{CRT}(S_p, S_q)$. Si supponga ora che un attaccante, agendo sul dispositivo di calcolo (smart-card), sia in grado di indurre un errore durante il calcolo di S_q, talché il risultato di questo calcolo sarà un numero *casuale* $r \in \mathbb{Z}_q$, in generale diverso da S_q. Denotiamo con S' la firma ottenuta, affetta da errore.

   (a) Dare la formula esplicita per S', quale si può dedurre dalla ricombinazione CRT.

   (b) Si supponga ora che l'attaccante possegga anche la firma corretta S. Mostrare come l'attaccante può facilmente recuperare p a partire $S - S'$ e da n.

   (c) Applicare l'attacco descritto nel punto precedente per ricavare p e q, nel caso di S, S' e n sotto descritti.

$$
\begin{aligned}
S &= 178750893623267289731765001502893928312011975692568578767 82 \\
S' &= 422560676697083921666351798843614917014922822120274939755935 \\
n &= 825500608838866132701444300844117841826444264266030066831623 \ .
\end{aligned}
$$

   Traccia. Per quanto riguarda (b), dalla formula CRT, si vede che $S - S' \bmod n = p(p^{-1} \bmod q)(S_q - r) \bmod n$. Dunque, assumendo senza sostanziale perdita di generalità che $r \neq S_q$, si vede che $(S - S', n) = p$. Più precisamente, applicando l'uguaglianza di cui sopra ed Euclide:

$$
\begin{aligned}
\big(S - S' \,,\, n\big) &= \big(n \,,\, S - S' \bmod n\big) \\
&= \big(n \,,\, p(p^{-1} \bmod q)(S_q - r) \bmod n\big) \\
&= \big(n \,,\, p(p^{-1} \bmod q)(S_q - r)\big) \\
&= p
\end{aligned}
$$

dove l'ultima uguaglianza deriva dal fatto che $q \not| (p^{-1} \bmod q)(S_q - r)$ (si noti in particolare che $0 < |S_q - r| < q$).

5. Siano m, n interi tali che $(m, n) = 1$. Dimostrare che, per ogni $x \in \mathbb{Z}_{mn}$, vale che $\text{ord}_{mn}(x) \geq \text{mcm}(\text{ord}_m(x), \text{ord}_n(x))$.

   Traccia. Sia $i = \text{ord}_{mn}(x)$. Vale dunque che $x^i \equiv_{mn} 1$. Proiettare questa uguaglianza su m e n usando il CRT. Deve allora valere che i è un multilplo di $\text{ord}_m(x)$ e di $\text{ord}_n(x)$ (perché?).

6. Si consideri un sistema di cifratura El Gamal con $p = 2579$, $\alpha = 2$ e $a = 765$. Il ricevente (proprietario della chiave privata) riceve il ciphertext $(\gamma, \delta) = (435, 2396)$. Decifrare (γ, δ) per ottenere il plaintext.

   Traccia. La risposta è $m = 1299$.

1.6 Funzioni hash crittografiche

1. Si diano le definizioni di funzione *weak* e *strong collision resistance* WCR/SCR.

   (a) Si discuta per ciascuna delle seguenti due funzioni se esse sono SCR o WCR.

   - $f : \mathbb{N} \to \mathbb{Z}_p$ definita come $f(x) \stackrel{\text{def}}{=} x \bmod p$, dove p è un primo grande pubblico;
   - $f : \mathbb{N} \to \mathbb{Z}_p^*$ definita come $f(x) \stackrel{\text{def}}{=} \alpha^x \bmod p$, dove p è un primo grande pubblico e α una radice primitiva di p, pubblica.

   (b) Si considerino le funzioni della forma $g : \mathbb{N} \to \mathbb{Z}_n^*$ definite da $f(x) \stackrel{\text{def}}{=} \alpha^x \bmod n$, dove $n = pq$ è il prodotto di due primi grandi segreti e $(\alpha, n) = 1$. Sia α che n sono pubblici. Si discuta se gli attacchi trovati nel caso (a) possono essere adattati a queste funzioni.

   Traccia. Nel caso (a), è facile per l'attaccante generare collisioni, cioè costruire coppie (x, x') con $x \neq x'$, tali che $f(x) = f(x')$: in particolare, per la seconda funzione, si applica il teorema di Fermat. Dunque le funzioni non sono WCR e dunque neanche SCR.

   Nel caso (b), il metodo impiegato nei due casi precedenti non funziona. In particolare, costruire una collisione (x, x') equivale a trovare $x' \neq x$ tali che $x \equiv_{\phi(n)} x'$ (Eulero): l'estensione degli attacchi precedenti richiede quindi la conoscenza di $\phi(n)$.

2. Sia $h : \mathcal{X} \to \mathcal{Y}$ una funzione che gode della proprietà di SCR. Supponiamo che $|\mathcal{X}| \geq 2|\mathcal{Y}|$. Si dimostri che h gode anche della proprietà di essere one-way. In particolare, si dimostri quanto segue: se esiste un algoritmo $A(y)$ che, dato $y \in \mathcal{Y}$, restituisce $x \in \mathcal{X}$ tale che $h(x) = y$ (attacco alla proprietà one-wayness), allora è possibile costruire un algoritmo probabilistico efficiente di tipo Las Vegas che viola la SCR con probabilità almeno $1/2$.

   Traccia. Ricordiamo che un algoritmo probabilistico di tipo Las Vegas restituisce o la risposta corretta, con probabilità superiore ad una certa soglia fissata (in questo caso $1/2$), oppure nessuna risposta. Sotto le assunzioni date, costruiamo un algoritmo Las Vegas B che restituisce una collisione per h, così descritto:

   (a) scegli $x \in \mathcal{X}$ uniformemente a caso e calcola $y = h(x)$ e $x' = A(y)$;

   (b) è vero che $x \neq x'$? Se si, restituisci (x, x').

E' chiaro che se B termina restituendo (x, x'), quest'ultima è una collisione per h (perché?). Qual è la probabilità che ciò avvenga? Per ogni $y_1, ..., y_m$ nell'immagine di h, poniamo $c_i = f^{-1}(y_i)$, la controimmagine di y_i secondo h. Chiaramente per ogni i, $c_i \subseteq \mathcal{X}$ e inoltre le $c_1, ..., c_m$ formano una *partizione* di $\mathcal{X}$. Inoltre, per ogni $i = 1, ..., m$, sarà $x_i = A(y_i)$ per un qualche $x_i \in c_i$. Se x al passo 1 dell'algoritmo viene scelto nella classe c_i di $\mathcal{X}$, la probabilità di successo di B sarà esattamente quella di *non* scegliere x_i:

$$\begin{aligned} \Pr(Succ|x \in c_i) &= \Pr(x \neq x_i | x \in c_i) \\ &= \frac{|c_i| - 1}{|c_i|}. \end{aligned}$$

La probabilità complessiva di successo per B sarà dunque:

$$\begin{aligned} \Pr(Succ) &= \sum_{i=1}^{m} \Pr(Succ|x \in c_i) \Pr(x \in c_i) \\ &= \sum_{i=1}^{m} \frac{|c_i| - 1}{|c_i|} \frac{|c_i|}{|\mathcal{X}|} \\ &= \sum_{i=1}^{m} \frac{|c_i| - 1}{|\mathcal{X}|} \\ &= \frac{|\mathcal{X}| - \sum_{i=1}^{m} 1}{|\mathcal{X}|} \\ &= \frac{|\mathcal{X}| - m}{|\mathcal{X}|} \\ &\geq \frac{|\mathcal{X}| - |\mathcal{Y}|}{|\mathcal{X}|} \\ &\geq \frac{|\mathcal{X}| - |\mathcal{X}|/2}{|\mathcal{X}|} \\ &= \frac{1}{2} \end{aligned}$$

dove si è usato anche il fatto che, naturalmente, $m \leq |\mathcal{Y}|$, cioè il numero di elementi dell'immagine di h non è superiore alla cardinalità di $\mathcal{Y}$.

3. Descrivere uno schema di autenticazione *Data Origin* che fornisca anche confidenzialità, in ciascuna delle due seguenti situazioni.

   (a) A e B condividono una chiave simmetrica K;

   (b) A e B non condividono una chiave simmetrica, ma A conosce una chiave pubblica di B, mentre B conosce una chiave di verifica (pubblica) di A. Lo schema dovrebbe fornire anche non-ripudiabilità ed evitare di applicare l'algoritmo di cifratura a chiave pubblica all'intero messaggio M.

   Traccia. Il primo caso è già stato ampiamente discusso e si basa sul fatto che la funzione di encryption a chiave condivisa $E_k[\cdot]$ goda di certe proprietà (effetto valanga). Nel secondo caso, si discuta lo schema:

$$A \to B : \; E_k[M], \; E_{K_B^+}\left[E_{K_A^-}[H(M)], k\right]$$

dove k è una chiave *one-time* generata casualmente da A. Dire quali sono le operazioni che B deve compiere per verificare il messaggio una volta ricevuto. Nel caso sia necessaria anche una prova che il messaggio è fresco, discutere una possibile modifica dello schema proposto sopra.

2 Esercizi di approfondimento

2.1 Probabilità di errore in Miller-Rabin

Lo scopo dell'esercizio è dimostrare che, nel caso in cui n è un numero composto dispari *non* Carmichael, sia il test di Fermat che di Miller-Rabin hanno probabilità di errore $\leq 1/2$ (per il caso in cui n è Carmichael, si rimanda alle *Note*, Appendice A.1. Si ricordi che questa limitazione superiore può essere migliorata a $\leq 1/4$, con una prova più sofisticata).

(a) Siano $x, y \in \mathbb{Z}_n^*$, dove x è un testimone di Fermat di n mentre y *non* è testimone di Fermat di n. Dimostrare che $xy \bmod n$ è un testimone di Fermat di n in $\mathbb{Z}_n^*$.

(b) Se x è un testimone di Fermat e $y, y' \in \mathbb{Z}_n^*$ sono due diversi non testimoni di Fermat, dimostrare che $xy \not\equiv_n xy'$.

(c) Dai due punti precedenti, concludere che se c'è almeno un testimone di Fermat (cioè n non è Carmichael), allora in $\mathbb{Z}_n^*$, per ciascuno dei non testimoni c'è almeno un testimone distinto. Dunque ci sono almeno $|\mathbb{Z}_n^*|/2 = \phi(n)/2$ testimoni di Fermat in $\mathbb{Z}_n^*$.

(d) Concludere che, se n è composto dispari non Carmichael, la probabilità che l'algoritmo ritorni *vero* è $\geq 1/2$.

2.2 Timing attack contro esponenziazione modulare (Kocher 1996)

Il tempo di esecuzione della moltiplicazione modulare tra due numeri di k bit, $a \times b \bmod n$, non è una costante che dipende solo da k, ma varia al variare dei due argomenti a e b. Di conseguenza, anche il tempo di esecuzione dell'algoritmo di esponenziazione per il calcolo di $c^d \bmod n$, una volta fissato l'esponente d di k bit, non è costante, ma varia al variare della base (ciphertext) c. In altre parole, si è visto sperimentalmente che, considerando diverse basi $c_1, c_2, ...$ scelte casualmente, i tempi ottenuti possono essere considerati come estrazioni i.i.d. da una variabile aleatoria T, che obbedisce ad una fissata distribuzione di probabilità. Denotando poi con T_j il tempo impiegato dall'iterazione j-ma ($j = k-1, ..., 0$) del ciclo `for`, si può porre

$$T = T_{k-1} + T_{k-2} + \cdots + T_0$$

dove le T_j sono a loro volta v.a. che obbediscono a una certa distribuzione di probabilità. Inoltre, si vede sperimentalmente che i tempi relativi a iterazioni diverse sono tra loro indipendenti. Nel calcolo di questi tempi, per semplicità, si tiene conto solo delle operazioni di moltiplicazione modulare, essendo trascurabili i tempi imputabili ad altre istruzioni (overhead per la gestione del ciclo `for`, assegnamenti etc.).

Queste caratteristiche possono essere sfruttate per condurre un attacco basato sull'osservazione ripetuta dei tempi di esecuzione dell'algoritmo, per diversi ciphertext c scelti a caso dall'attaccante. L'attacco funziona con tutte le varianti note dell'algoritmo. Esso verrà qui illustrato nel caso dell'algoritmo da sinistra a destra, discusso a lezione.,

(a) Si ricordi che la *varianza* di una variabile aleatoria a valori reali X di media μ, quando esiste finita, è data da: $\mathrm{var}(X) \stackrel{\text{def}}{=} E[(X-\mu)^2] = E[X^2] - \mu^2$. Provare che per due variabili aleatorie indipendenti X e Y, vale $\mathrm{var}(X+Y) = \mathrm{var}(X) + \mathrm{var}(Y)$.

(b) Si assuma che l'attaccante conosca già i bit da d_{k-1} a d_{i+1} (compresi) dell'esponente. Egli vuole ora dedurre il bit d_i ($k-1 \geq i \geq 0$). Egli ipotizza un valore $d'_i \in \{0,1\}$ per d_i e, servendosi di una copia identica del dispositivo (smart-card) da attaccare, emula l'algoritmo ***fino all'iterazione di indice*** i**-mo inclusa**. Egli annota il tempo impiegato da tale esecuzione, $T' = T'_{k-1} + \cdots + T'_{i+1} + T'_i$. Confronta poi tale tempo con il tempo di esecuzione T del dispositivo che esegue ***l'operazione completa***, con la medesima base. Dare una formula per $T - T'$.

(c) Dare una formula per $\mathrm{var}(T-T')$ nei due casi: $d'_i = d_i$ (valore ipotizzato corretto), $d'_i \neq d_i$ (valore ipotizzato scorretto). Nel caso che $d_i \neq d'_i$, le variabili T_i e T'_i possono essere considerate (approsimativamente) indipendenti. Si assuma per semplicità che le T_j e le T'_j abbiano tutte la stessa varianza v.

(d) Supponiamo che l'attaccante ripeta l'operazione del punto (b) due volte: prima ipotizzando $d'_i = 0$, e poi che $d'_i = 1$. Formulare una regola che permetta all'attaccante di stabilire se $d_i = 0$ o $d_i = 1$.

(e) Dire come può essere stimata, in pratica, $\mathrm{var}(T-T')$, in base alla legge debole dei grandi numeri.

(f) Concludere che l'attaccante può dedurre i bit dell'esponente uno alla volta con la tecnica sopra esposta, partendo dal bit più significativo (ponendo cioè all'inizio $i = k-1$).

(g) Si riconsideri il punto (c): perché al posto della varianza non può semplicemente essere usata la media di $T - T'$?

Traccia. Per il punto (b), si osservi che deve essere $T'_j = T_j$ per $j = k-1, ..., i+1$. Per (c), ci si rifaccia all'indipendenza tra le T_j e al punto (a): porre attenzione a valutare correttamente $\mathrm{var}(T_i - T'_i)$, nel caso $d'_i \neq d_i$. Per (d), si chiamino T'^0 e T'^1 i tempi parziali ottenuti ponendo $d'_i = 0, 1$ rispettivamente, nell'esperimento del punto (b): il valore corretto di d_i corrisponde alla varianza minore tra $\mathrm{var}(T-T'^0)$ e $\mathrm{var}(T-T'^1)$ (perché?). Per (e), si tenga presente che la varianza è un valore atteso, e dunque può essere stimata come una media aritmetica, usando N estrazioni i.i.d. di $T - T'$, per N abbastanza grande:

$$\mathrm{var}(T-T') \approx \frac{1}{N}\sum_{j=1}^{N}(t^{(j)} - t'^{(j)})^2 - \left(\frac{1}{N}\sum_{j=1}^{N}(t^{(j)} - t'^{(j)})\right)^2 .$$

Le $t^{(j)} - t'^{(j)}$ rappresentano qui estrazioni indipendenti di $T - T'$, ciascuna ottenuta misurando i tempi di esecuzione T e T' relativi ad un ciphertext (base) c_j generato casualmente dall'attaccante, per $j = 1, ..., N$.

Per maggiori informazioni sull'attacco, consultare per esempio la voce Wikipedia su *Timing attack*, e i riferimenti ivi contenuti:
`https://en.wikipedia.org/wiki/Timing_attack`.

2.3 Funzioni hash crittografiche resistenti

Vogliamo stabilire alcune semplici limitazioni circa lo sforzo necessario (o sufficiente) per attaccare delle funzioni hash crittografiche resistenti. In questo tipo di ragionamenti, il risultato dell'applicazione di una funzione hash $H(\cdot)$ ad un messaggio, viene modellizzato come l'estrazione da una variabile aleatoria uniformemente distribuita su $\mathcal{Y} = \{0,1\}^m$. Più estrazioni vengono considerate come indipendenti. Questo modellizza, in maniera probabilistica, il fatto che l'output di una funzione hash è difficile da prevedere. Per semplicità, questo modello ignora il fatto che se un medesimo input occorre due o più volte, esso produrrà lo stesso output tutte le volte. Questo tipo di modellizzazione si può applicare anche al caso in cui l'input della funzione è parzialmente specificato come un blocco S fissato e noto, cioè quando si considera la funzione $H(S,\cdot)$ anziché $H(\cdot)$.

(a) (Attacco a one-wayness) Dato un digest $y_0 \in \mathcal{Y}$, dare una formula per la probabilità che una singola estrazione di H produca il digest y_0, cioè che $\Pr(H = y_0)$. Detto N il numero di estrazioni indipendenti fino alla prima occorrenza di y_0 inclusa, dare una formula per il valore atteso di N, cioè $E[N]$.

(b) (Attacco a one-wayness, caso generale) Dato un insieme di digest $I = \{y_1, ..., y_k\} \subseteq \mathcal{Y}$, dare formule per $\Pr(H \in I)$ e per $E[N_I]$, dove N_I = n. di estrazioni indipendenti fino alla prima occorrenza inclusa di un $y \in I$. Data una funzione hash con message digest di $m = 160$ bit, stimare quante estrazioni sono mediamente necessarie per produrre un digest i cui i 20 bit più significativi siano tutti 0. Questo tipo di ragionamento è alla base di alcuni sistemi *proof-of-work*, come quelli usati in HashCash e nel Bitcoin mining. L'idea di base è che, a partire da un seed S specificato da un utente verificatore, l'utente che vuole inviare una mail (HashCash) o coniare dei nuovi bitcoin (mining), debba produrre una coppia (M, y) tale che $H(S, M) = y$ e il digest y è della forma specificata (es. 20 cifre più significative a 0). Si noti che per il verificatore è facile controllare, dato y e M, se $H(S, M) = y$.

(c) Dire se e in che modo le considerazioni precedenti si estendono agli attacchi alla weak collision resistance.

(d) Data una sequenza di n estrazioni i.i.d. $H_1, H_2, ..., H_n$, e una coppia di indici $\{i, j\}$, con $i \neq j$, una *collisione tra i e j* è l'evento $(H_i = H_j)$. Fissati $i \neq j$ e chiamata Y_{ij} la variabile indicatrice che dà 1 se c'è collisione tra i e j, 0 altrimenti, dare delle formule per $\Pr(Y_{ij} = 0)$ e $E[Y_{ij}]$.

(e) Chiamiamo C la v.a. che dà numero totale di collisioni nella sequenza $H_1, H_2, ..., H_n$. Provare che $C = \sum_{(i,j):i<j} Y_{ij}$. Sfruttando la linearità del valore atteso $E[\cdot]$ e il punto precedente, dare quindi una formula per $E[C]$ che dipenda da n e da $|\mathcal{Y}|$.

(f) (Attacco a strong collision resistance) Provare che se n è almeno $\approx \sqrt{2|\mathcal{Y}|}$ allora $E[C] \geq 1$. Questa è una versione del *paradosso del compleanno*. Applicare questo risultato, dando una stima di quante persone è necessario far entrare in una stanza, per avere mediamente almeno una collisione, cioè due persone che compiono il compleanno lo stesso giorno (in questo caso $\mathcal{Y}$ = giorni dell'anno; si assume che i compleanni siano uniformemente distribuiti su tutti i giorni dell'anno).

(g) Si consideri la stringa s dei 12 bit meno significativi del proprio numero di matricola, rappresentato in binario. Come caso particolare dell'attacco (a), creare un messaggio

m tale che i 12 bit meno significativi del digest $SHA1(m)$ coincidano con s. Riportare il messaggio m (in esadecimale), il numero di tentativi e il tempo impiegato dal PC a trovare tale m. NB: sulle librerie di funzioni hash in Python, si veda: `https://docs.python.org/3/library/hashlib.html`.

(h) Si implementi un attacco del compleanno sulla funzione $SHA1$ limitata ai 12 bit meno significativi del message digest. Si riporti la collisione (m, m') trovata, il numero di coppie generate e il tempo impiegato, e li si confronti con il caso (g).

2.4 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 $K_1^+ = \langle 3, n\rangle$ e $K_2^+ = \langle 11, n\rangle$. Un secondo utente invia ad A, in tempi diversi, lo stesso messaggio m, cifrato prima con la chiave K_1 e poi con la chiave K_2. Un attaccante intercetta i relativi plaintext, $c_1 = E_{K_1^+}[m]$ e $c_2 = E_{K_2^+}[m]$, che numericamente valgono

$$\begin{aligned} c_1 &= 415459980059712388764580516278528357540868548132004893964 33 \\ c_2 &= 884141165346707443294744910953393011210663087557694028365 77\,. \end{aligned}$$

Ricavare m a partire dalle informazioni disponibili, *senza* fattorizzare n o ricavare gli esponenti privati. Illustrare i vari passaggi.

3 Esercizi di programmazione

Per gli esercizi di seguito proposti, si intende che la soluzione deve essere programmata nel linguaggio Python.
NB: gli esercizi di programmazione svolti e consegnati ai fini dell'esonero DEVONO consistere di: (1) una parte che risponde in maniera sintetica ma esauriente alle domande poste nel testo dell'esercizio e illustra le tecniche e gli algoritmi usati; (2) il codice vero e proprio, corredato di istruzioni per l'uso e commenti, che può essere inviato via email al docente.

3.1 Implementazione di algoritmi per crittografia a chiave pubblica

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 ($> 10^{100}$). 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).

3.2 Attacco al decryption exponent in RSA

Lo scopo dell'esercizio è dimostrare *sperimentalmente* che, in uno uno schema RSA con un modulo $n = p \cdot q$, se l'esponente di decryption d è noto all'attaccante, egli può facilmente fattorizzare n. L'attacco è costituito dal seguente algoritmo probabilistico di tipo Las Vegas.

Supponiamo che l'attaccante sia entrato in possesso di $d \in \mathbb{Z}_n^*$ t.c. $de \equiv_{\phi(n)} 1$, cioè $ed = k\phi(n) + 1$, per un certo intero k, dove e è l'esponente pubblico (anch'esso noto).

1. Porre $ed - 1 = 2^r \cdot m$, per m dispari e $r \geq 1$.
2. Scegliere $x \in \mathbb{Z}_n^*$ uniformemente a caso, e calcolare i quadrati successivi: $x_0 = x^m \bmod n, x_1 = x^{2m} \bmod n, ..., x_r = x^{2^r m} \bmod n \equiv_n x^{ed-1}$.
3. Sia j il minimo indice tale che $x_j = 1$. Se $j > 0$ e $x_{j-1} \not\equiv_n -1$, allora restituire il $\mathrm{MCD}(x_{j-1} + 1, n)$ e stop. Altrimenti ripetere dal punto 2.

Si può dimostrare che il numero medio di iterazioni eseguite dall'algoritmo è ≤ 2.

(a) Spiegare perché l'indice j di cui al passo 3 dell'algoritmo esiste sicuramente e perché, quando termina, l'algoritmo fornisce un fattore non banale di n.

(b) Programmare una funzione Python decryptionexp(n,d) che implementa l'attacco, dove n è un modulo RSA e d un suo esponente di decryption. Testare quindi la funzione su 100 moduli RSA generati casualmente, ma realistici ($n > 10^{100}$), riportando in una tabella il n. medio di iterazioni dell'algoritmo, il tempo medio di esecuzione dell'attacco e la varianza di tale tempo (NB: si intende escluso il tempo per la generazione dei moduli n e degli esponenti d).

3.3 Timing attack

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 $c^d \bmod n$.

- 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`).