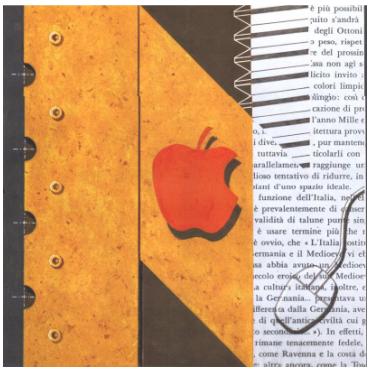


Newton-Pertini di Camposampiero

Crittografia

Student :
Alex Gasparini

Teacher :
Gianfranco Lamon

31 luglio 2025

Indice

Introduzione In questo corso vedremo le basi della **crittografia**, analizzando anche minuziosamente alcuni aspetti. La crittografia è quella branca della matematica e dell'informatica che studia come inviare messaggi e informazioni in modo che non siano comprensibili o leggibili da chi non è il mittente o il destinatario del messaggio. Inizieremo da una crittografia "semplice": **One Time Pad** (spesso abbreviato in **OTP**), con "semplice" intendo dire che per comprenderlo non sono necessarie conoscenze preliminari. D'altro canto, le altre tecniche che vedremo si basano su una solida base matematica, molto complessa, ma riprenderemo anche queste nozioni matematiche. In particolare, esamineremo nel dettaglio gli algoritmi **RSA** e **Diffie-Hellman**, che hanno due funzionalità ben distinte ma sono comunque pietre miliari della crittografia moderna. In entrambe le tecniche analizzeremo anche le dimostrazioni e forniremo esempi pratici per comprendere a fondo questo argomento, che, pur essendo apparentemente difficile, affascina per le sue applicazioni nella vita reale.

1 Crittografia: introduzione e prime definizioni

2 Cifrati Antichi - Cifrario di Cesare

3 Cifrati Antichi - Enigma

La tecnica **OTP**, per come l'abbiamo studiata fino ad ora, è utilizzabile solo tramite i computer, ma in realtà è molto più antica e, durante la **Seconda Guerra Mondiale**, ha giocato un ruolo fondamentale. Infatti, i tedeschi hanno utilizzato la **Macchina Enigma**, che permetteva loro di comunicare inviando messaggi cifrati. La macchina Enigma, per l'appunto, usa una tecnica di crittografia molto simile, se non uguale, all'**OTP**.

Foto della macchina Enigma

3.1 Preambolo Storico

La macchina Enigma è stata brevettata da **Arthur Scherbius**, un ingegnere tedesco, nel **1918**. Al contrario di quanto si pensa, Enigma veniva utilizzata anche prima della Seconda Guerra Mondiale, soprattutto dalla **marina militare tedesca**. Tuttavia, Enigma ha avuto un ruolo importantissimo durante la Seconda Guerra Mondiale, perché ha permesso ai tedeschi di inviare messaggi sicuri, impedendo agli Alleati di comprendere le decisioni militari tedesche.

La macchina Enigma aveva una **chiave privata** (che vedremo in dettaglio più avanti) che veniva cambiata **ogni giorno a mezzanotte**. La chiave veniva inviata soltanto agli ufficiali tedeschi di grado più alto per evitare fughe di informazioni riservate. Infatti, verso mezzanotte, ai generali tedeschi arrivava un messaggio chiamato **Schlüsselheft**, che conteneva la chiave del giorno. In realtà, alcuni reparti tedeschi, come la marina militare, che richiedevano maggiore sicurezza, cambiavano chiave ogni **8 ore**. Inoltre, erano previsti protocolli di sicurezza, come il fatto che il **Schlüsselheft** dovesse essere conservato in un luogo sicuro sotto chiave, con accesso riservato solo a poche persone autorizzate. Se vi era il rischio che gli Alleati potessero entrarne in possesso, il **Schlüsselheft** doveva essere immediatamente distrutto tramite il fuoco.

3.2 Funzionamento

La macchina Enigma serviva per crittografare e decrittografare i messaggi. Era dotata di una tastiera e, alla pressione di un tasto (dove ogni tasto rappresentava una lettera dell'alfabeto tedesco), si accendeva un led che indicava un'altra lettera dell'alfabeto. Per utilizzare una macchina Enigma, di solito erano necessarie due persone: una che scriveva il messaggio da cifrare o decifrare e un'altra che annotava le lettere che si illuminavano.

Nell'immagine possiamo notare che, alla pressione del tasto **I** sulla tastiera, si illumina la lettera **U** nella parte superiore.

3.3 Struttura

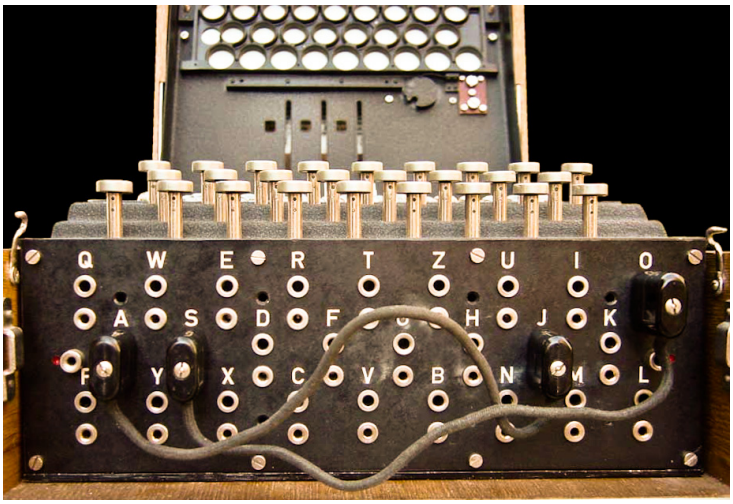
Ogni macchina Enigma era composta da sei parti principali: **Tastiera**, **Rotori**, **Riflettore**, **Lampboard**, **Plugboard** e la **Batteria**.

Partiamo dagli elementi più semplici: la batteria serviva ad alimentare la parte elettrica della macchina, mentre la tastiera conteneva le 26 lettere dell'alfabeto sotto forma di pulsanti. La tastiera funzionava in modo simile a quelle odierne, nel senso che, alla pressione di un tasto, si chiudeva il circuito elettrico.

La **Lampboard** (detta **Lampenbrett** in tedesco) era la parte in cui si trovavano le lettere che si illuminavano alla pressione dei tasti sulla tastiera.

3.3.1 Plugboard

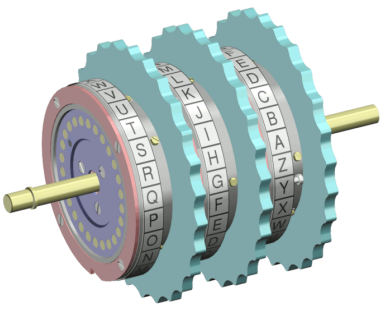
La **Plugboard** (detta anche **Steckerbrett**) serviva per invertire una coppia di lettere. Questa inversione avveniva sia in input (quindi prima di eseguire l'algoritmo di cifratura) sia in output (dopo aver cifrato il messaggio). Questa caratteristica serviva esclusivamente ad aggiungere un ulteriore livello di sicurezza, ma non era il punto focale della macchina.



Nell'immagine possiamo notare come siano collegate le coppie (**A**, **J**) e (**S**, **O**). Quindi, se veniva premuto il tasto **A** sulla tastiera, esso veniva prima trasformato in **J**, che poi seguiva il resto dell'algoritmo. Allo stesso modo, se la lettera crittata era **S**, questa veniva cambiata in **O**, e quindi si illuminava la lettera **O** sulla lampboard.

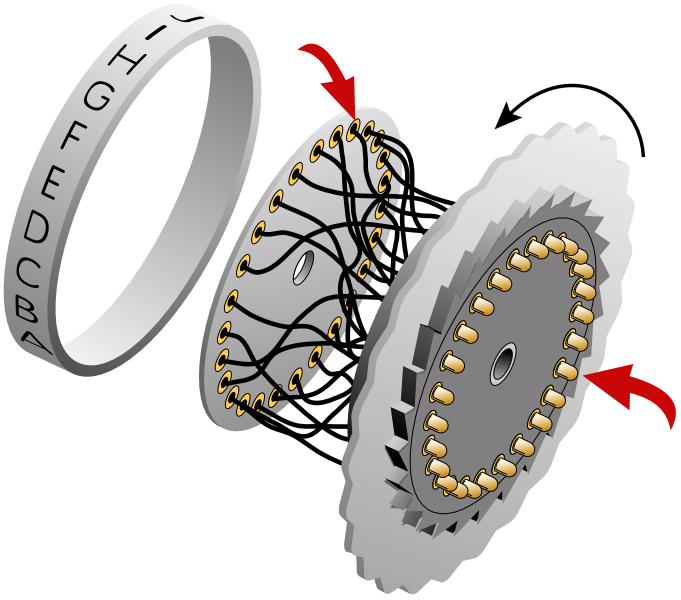
3.3.2 Rotori

La parte principale dell'algoritmo di cifratura sono **i rotori**, che permettono di trasformare le lettere per crittarle. Un rotore è un componente metallico circolare collegato a un ingranaggio che gli permette di ruotare e cambiare lettera, come vedremo in seguito. Inoltre, è montato su un asse che ne consente la rotazione e, su entrambi i lati, presenta 26 punti di contatto metallici che permettono di trasmettere la corrente tra un rotore e l'altro.



Render 3D di tre rotori collegati

Per comprendere meglio il funzionamento di un rotore, è utile analizzarne la struttura interna:



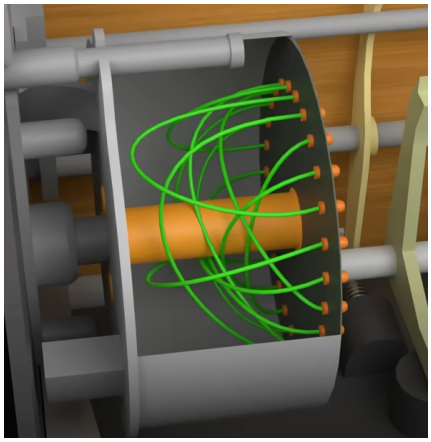
Come accennato in precedenza, entrambi i lati del rotore presentano piastre metalliche per la conduzione della corrente (indicate dalla freccia rossa nell'immagine). Inoltre, ogni punto di contatto su un lato è collegato a un punto sull'altro lato, ma in modo disordinato. Questo collegamento permette di mescolare le lettere: una lettera che "entra" nel rotore ne "uscirà" come un'altra, a causa di questo rimescolamento.

Di norma, una macchina Enigma contiene tre rotori collegati in serie per aumentare la complessità della cifratura. Tuttavia, alcune versioni della macchina Enigma utilizzano quattro o più rotori, semplicemente per incrementare la sicurezza e rendere ancora più difficile la decrittazione.

3.3.3 Riflettore

Per aumentare ulteriormente la sicurezza, i tre rotori sono collegati a un **Riflettore**, il quale fa "rientrare" il segnale nei rotori, permettendo un'ulteriore trasformazione delle lettere. Questo avviene perché la corrente attraversa nuovamente tutti e tre i rotori in senso inverso.

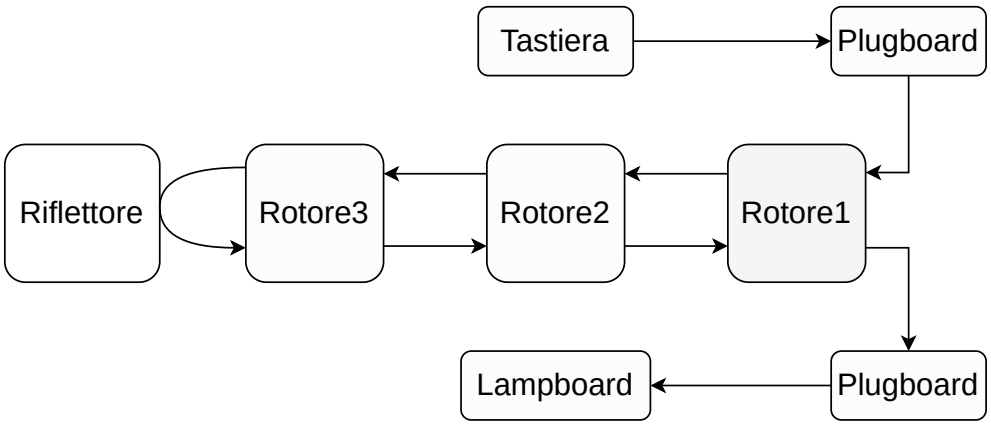
In questo modo, una lettera viene cifrata **sei volte** prima di essere visualizzata sulla lampboard.



Render 3D di un Riflettore

3.4 Funzionamento

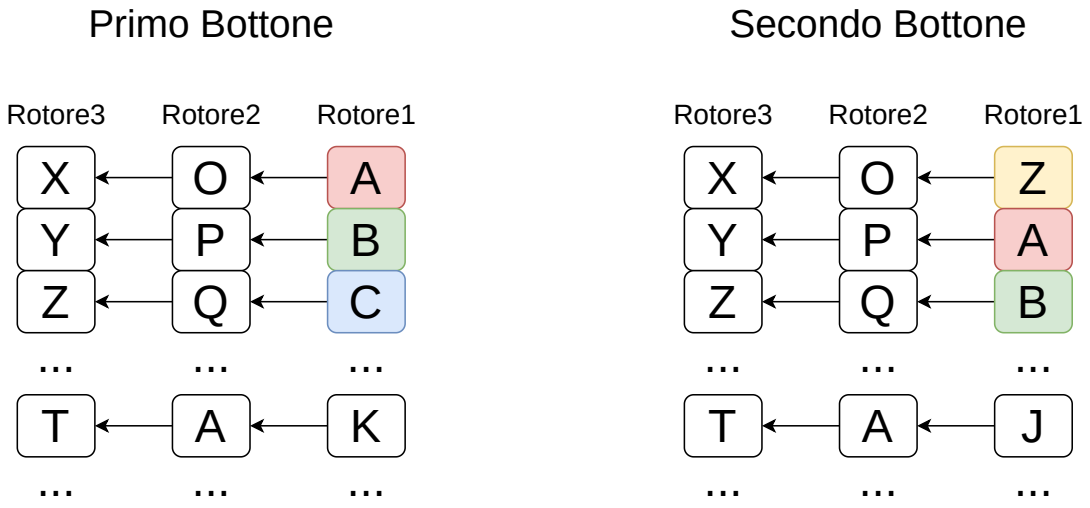
Ripercorriamo il funzionamento della macchina Enigma.



Quando viene premuto un tasto sulla **Tastiera**, il segnale passa attraverso la **Plugboard**, che sostituisce la lettera con quella associata se un cavo è collegato; altrimenti, il segnale prosegue direttamente verso i **Rotori**. Successivamente, attraversa i tre rotori, modificando la lettera tre volte. Grazie al **Riflettore**, il segnale viene reinviato attraverso i rotori, che trasformano nuovamente la lettera per altre tre volte. Infine, il segnale passa nuovamente per la **Plugboard**, che può applicare un'ulteriore sostituzione se la lettera è collegata a un'altra. Dopo tutte queste trasformazioni, la lettera finale viene visualizzata sulla **Lampboard**.

Durante questo processo, la lettera viene trasformata un minimo di **6 volte** e fino a un massimo di **8 volte** se la **Plugboard** modifica sia l'input che l'output.

Non appena il tasto premuto viene rilasciato, il primo rotore ruota di una posizione grazie a un ingranaggio, modificando così la configurazione delle lettere. Questo meccanismo assicura che, anche premendo due volte di seguito la stessa lettera, vengano visualizzate lettere completamente diverse, poiché la rotazione del rotore cambia continuamente le combinazioni possibili.



4 One Time Pad

La prima forma di crittografia che vediamo è **One Time Pad**, che è una crittografia **Simmetrica**. Con il termine crittografia simmetrica si intendono tutte le tecnite crittografiche che usano una sola chiave per criptare e decriptare, al contrario della crittografia asimmetrica che ne usa due, ma questo lo vedremo bene più avanti. Prima di capire bene il funzionamento di questa tecnica è meglio rivedere cos'è lo **XOR** e come funziona.

4.1 XOR

Lo **XOR** (*eXclusive OR*) è un operatore booleano binario la cui tabella di verità è la seguente:

A	B	XOR(A, B)
0	0	0
0	1	1
1	0	1
1	1	0

Per ricordarla a memoria, basta sapere che, se A e B sono diversi, lo XOR restituisce 1; altrimenti, se sono uguali, restituisce 0.

Ci interessano in particolare le seguenti proprietà dello XOR:

$$(A \oplus B) \oplus B = A$$

$$A \oplus 0 = A$$

dove $\oplus$ è il simbolo dello XOR. In realtà, queste proprietà sono tutto ciò che ci serve per comprendere il nostro algoritmo crittografico, quindi passiamo ad analizzarne il funzionamento.

4.2 Funzionamento

L'algoritmo **OTP** inizia generando una chiave lunga almeno quanto il messaggio da cifrare. La chiave può assumere qualsiasi forma, purché sia privata e nessuno la conosca. Dopo la generazione della chiave, è sufficiente XORarla con il messaggio che vogliamo inviare:

$$C = M \oplus K$$

dove C è il messaggio cifrato (*Crypted message*), M è il messaggio originale (*Message*) e K è la chiave (*Key*).

Una volta inviato il messaggio, il destinatario può semplicemente applicare l'operazione XOR tra il messaggio ricevuto e la chiave per recuperare il messaggio originale:

$$C \oplus K$$

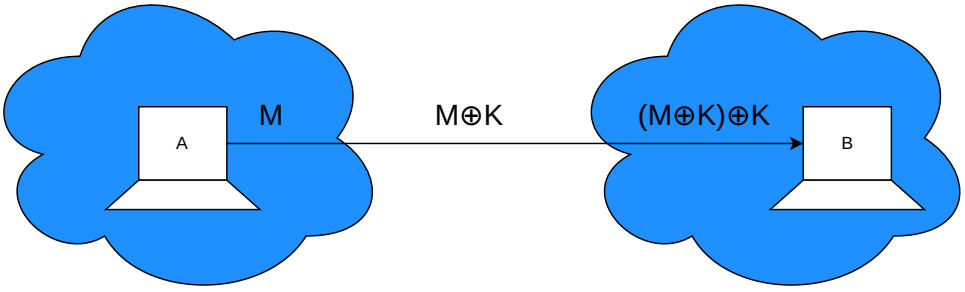
Per capire perché il metodo funziona, sostituiamo l'equazione precedente ($C = M \oplus K$):

$$(M \oplus K) \oplus K$$

Utilizzando la proprietà dello XOR, otteniamo:

$$M = (M \oplus K) \oplus K$$

In questo modo, applicando l'operazione XOR, recuperiamo esattamente il messaggio originale.



4.2.1 Considerazioni

Questo algoritmo, però, non si occupa della condivisione sicura della chiave. Per questo scopo si può utilizzare l'algoritmo **Diffie-Hellman**, che è progettato proprio per condividere chiavi private in modo sicuro.

La forza e la bellezza di questo algoritmo risiedono nel fatto che è **completamente sicuro**: è stato infatti dimostrato matematicamente che è impossibile decifrare il messaggio cifrato ($M \oplus K$) senza conoscere la chiave. L'**OTP** è, infatti, l'unico algoritmo **impossibile da decifrare** e completamente sicuro che conosciamo al momento. L'unico metodo per indovinare il messaggio è tentare casualmente e sperare in un colpo di fortuna.

Ovviamente, non è tutto oro ciò che luccica: se fosse davvero perfetto, useremmo solo questa tecnica e saremmo al 100% sicuri. Tuttavia, come vedremo, questo algoritmo presenta un problema non da poco.

4.3 Crackabilità

Il problema di questa tecnica risiede nel nome: **ONE TIME Pad**. Ovvero, questa tecnica si può usare **una sola volta** con la stessa chiave. Questo è un requisito fondamentale, perché qualora una chiave venga usata più di una volta, tramite varie tecniche si può estrapolare la chiave e i messaggi, almeno parzialmente.

Questo accade perché, dati due messaggi in chiaro ($\mathbf{M_1}$, $\mathbf{M_2}$), una chiave privata ($\mathbf{K}$) e le loro combinazioni cifrate ($\mathbf{C_1}$, $\mathbf{C_2}$):

$$C_1 = M_1 \oplus K$$
$$C_2 = M_2 \oplus K$$

Supponiamo di aver intercettato i messaggi cifrati: possiamo applicare l'operazione XOR tra di loro:

$$C_1 \oplus C_2$$

Riscriviamo l'espressione sostituendo i valori di C_1 e C_2:

$$C_1 \oplus C_2 = (M_1 \oplus K) \oplus (M_2 \oplus K)$$

Ora, grazie alla proprietà associativa dello XOR, possiamo riorganizzare i termini:

$$C_1 \oplus C_2 = (M_1 \oplus M_2) \oplus (K \oplus K)$$

Poiché $K \oplus K = 0$, otteniamo:

$$C_1 \oplus C_2 = (M_1 \oplus M_2) \oplus 0$$

Infine, ricordando che $X \oplus 0 = X$, il risultato finale è:

$$C_1 \oplus C_2 = M_1 \oplus M_2$$

Dopo tutti questi passaggi, abbiamo scoperto che la XOR tra i due messaggi cifrati è uguale alla XOR dei due messaggi in chiaro. In questo modo, siamo riusciti a "rimuovere" la chiave. Chiaramente, con la XOR dei messaggi in chiaro, comunque non riusciamo a decriptarli, ma possiamo dedurre alcune parti. Infatti, esistono alcuni algoritmi, come **crib drag**, che, dati alcuni messaggi cifrati, possono decifrare parti dei messaggi e della chiave. Questi algoritmi sono particolarmente complessi e sfruttano certi pattern nelle frasi (come banalmente gli spazi, oppure gli articoli nelle varie lingue) e anche degli **attacchi con dizionario** (dictionary attack). Per capirli bene e in maniera pratica, consiglio di scaricare i seguenti script Python: https://github.com/CameronLonsdale/MTP. Inoltre, per gli amanti del rap, vi lascio un esempio divertente per cercare di capire quale canzone è: Link Drive.

5 Cifrari a Blocchi - DES

6 Cifrari a Blocchi - AES

7 RSA

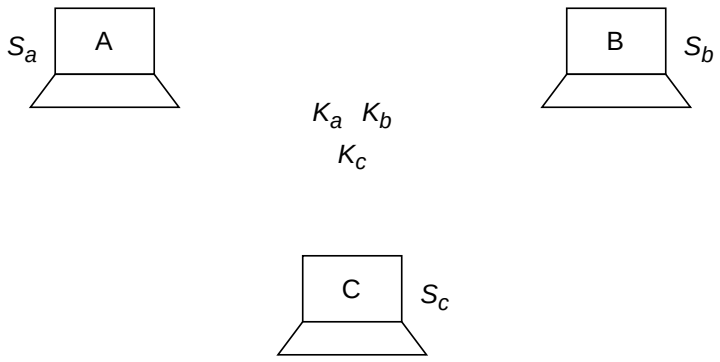
Nel 1977 i crittografi **Ronald Rivest**, **Adi Shamir** e **Leonard Adleman** inventarono l'algoritmo **RSA** (dove RSA è l'acronimo dei cognomi dei 3 crittografi). L'RSA è un algortimo per generare una coppia di chiavi private e pubbliche, quindi per definizione rientra nella crittografia asimmetrica. RSA si occupa solo di generare le chiavi, ma non si occupa di condividere privatamente la chiave privata, per quello vedremo l'algoritmo di **Diffie-Hellman**.

7.1 Funzionamento Generale

Partiamo capendo come funziona l'algoritmo da un punto di vista logico. Gli algoritmi di crittografia si divisono in due principali gruppi: **Simmetrica** e **Asimmetrica**. La crittografia simmetrica sfrutta una solo chiave, usata sia per criptare che decriptare un messaggio, che chiaramente deve essere privato perchè senno può essere decifrato da chiunque. La crittografia asimmetrica invece usa 2 chiavi: **Publica**, **Privata**. Quella publica è usata per criptare un messaggio, ed è detto pubblico perchè chiunque può averla, mentre la chiave privata serve per decifrare il messaggio (chiaramente va tenuta segreta e non va distribuita).

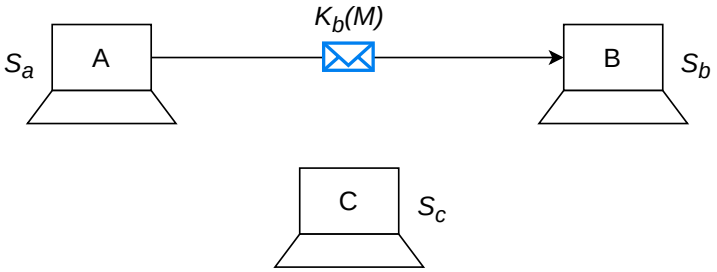
L'RSA è un sistema crittografico asimmetrico, quindi posside una chiave privata e una publica. L'algoritmo è prettamente matematico e pre funzionare usa teoremi matematici, infatti le chiavi non sono altro che combinazioni di numeri, vedremo dopo che la chiave privata è definita come (**n**, **d**), mentre la chiave publica è definita (**n**, **e**) dove **n**,**d**, **e** sono tutti numeri.

Facendo un esempio pratico: supponiamo che abbiamo una rete con 3 computer (chiamiamoli come A, B, C), ciascun dispositivo ha chiaramente una propria chiave publica e una privata, denominiamo K_a, K_b, K_c le chiavi publiche e S_a, S_b, S_c le chiavi private (S sta per *secret*)



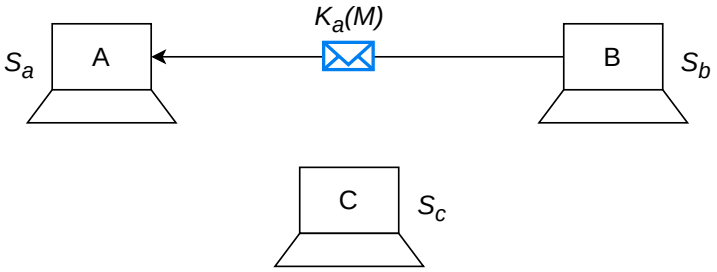
Ogni dispositivo conosce la chiave publica (**K**), mentre tiene per sè la propria chiave publica (**S**)

Supponiamo che il **PC A** vuole inviare al **PC B** un pacchetto qualsiasi. Allora il PC A prenderà la chiave publica di B (K_b), con quella chiave cripta il messaggio che vuole mandare e lo invierà al PC B.



Con M si intende il messaggio da inviare e con $K_b(M)$ è il messaggio criptato con la chiave K_b.

In questa maniera se C provasse a intercettare il messaggio non potrà leggere il contenuto perchè è criptato con la chiave publica di B. Solamente B protrà decriptarlo con la sua chiave privata. Se B dovrà rispondere ad A, allora cripterà il messaggio con la chiave publica di A, in modo che solo A protrà decriptarlo.



Un problema che può nascere da questo algoritmo è se le chiavi publiche sono falsificate. Infatti può succedere che il PC C invia a tutti i una chiave publica dicendo che è quella del PC A, quando invece è la sua. Facendo così i PC che vogliono comunicare con il PC A cripteranno i messaggi con la chiave publica di C (perchè è stata falsificata), e che quindi PC C potrà leggere il messaggio. Per evitare questo problema si usa un altro protocollo, **Diffie-Hellman** per iniviare le chiavi publiche avendo la certezza che siano del PC corretto.

7.2 Basi di Matematica Modulare

L'algoritmo RSA è fortemente basato su dei fondamentali di aritmetica modulare molto complessa, quindi prima di descrivere l'algoritmo è meglio ricordare dei principi fondandi di tale algoritmo.

7.2.1 Aritmetica Modulare

In primis rivediamo cosa è l'aritmetica modulare. L'aritmetica modulare è quella branca della matematica che studia le operazioni in **modulo**. Per fare un esempio dobbiamo rivedere come facevamo le divisioni alle elementari, ovvero con il resto. Facciamo la divisione tra 7 e 2.

$$\frac{7}{2} = 3 \ \textit{resto} \ 1$$

In aritmetica modulare questa divisione la possiamo scrivere anche nel seguente modo

$$7 \equiv 1 \mod 2$$

Si legge "7 è congruo a 1 in modulo 2". Questo vuol dire che se dividiamo 7 per 2 e anche 1 per 2 avranno lo stesso resto (cioè 1). Infatti possiamo dire anche che

$$7 \equiv 3 \mod 2$$

Perche anche 3 diviso 2 da come resto 1. Capiamo che con le operaioni modulari a noi interessa soltanto il resto della divisione tra i 2 numeri. Infatti se vogliamo generalizzare, dalla congruenza seguente

$$a \equiv b \mod n$$

possiamo affermare che

$$a = k * n + b \tag{1}$$

dove k è un qualsiasi numero, per esempio con la congruenza $7 \equiv 3 \mod 2$ possiamo riscriverla come $7 = 2*k+3$, ed in questo caso $k = 2$. Questa proprietà sarà fondamentale per il funzionamento dell'algoritmo

7.2.2 Teoremi Fondamentali

Per proseguire dobbiamo definire cosa vuole dire che 2 numeri sono **coprimi**. Due numeri si dicono coprimi se non hanno nessun divisore in comune (apparte 1). In altre parole che il loro massimo comun divisore è 1. Con questa definizione possiamo costruire un insieme molto particolare: Z_n^* (si legge "Zeta n star") dove n è un numero intero positivo. Dato un numero n, Z_n^* contiene tutti i numeri minori di n coprimi ad n. Per fare chiarezza prendiamo un numero, ad esempio $n = 15$. Prendiamo tutti i numeri minori di 15 e scegliamo soltanto quelli che non hanno divisori in comune con 15.

$$Z_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$$

Da notare che se n è un numero primo, allora l'insieme Z_n^* sono tutti i numeri interi positivi minori di n. Ad esempio se $n = 7$ allora

$$Z_7^* = \{1, 2, 3, 4, 5, 6\}$$

In realtà più che l'insieme in sè a noi interessa quanti numeri ci sono al suo interno. Per farlo c'è una funziona particolare, chiamata **Funzione Toziente di Eulero**, che dato un numero riesce a calcolare quanti numeri sono coprimi a quel numero. Detto in altre parole, la funzione conta quanti elementi ci sono nell'insieme Z_n^*. La funzione viene anche detta **Funzione ϕ di Eulero** perchè viene indicata così $\phi(n)$.

Con l'esempio che abbiamo visto prima, sappiamo che $\phi(15) = 8$, perche se andiamo a contare quanti elementi contiene Z_{15}^* fa proprio 8. Per ora quindi per calcolare La funzione ϕ di un numero, prima dobbiamo trovare tutti i numeri coprimi al numero (quindi trovare Z_n^*) e calcolare quanti sono i numeri. Però c'è un modo molto più veloce per calcolarlo. Per calcolarlo infatti possiamo scomporre il numero a fattori primi (indicati con la lettera p)

$$n = p_1 * p_2 * ... * p_k$$

allora

$$\phi(n) = (p_1 - 1) * (p_2 - 1) * ... * (p_k - 1)$$

Tornando al solito esempio di $n = 15$

$$15 = 3 * 5$$

allora

$$\phi(15) = (3 - 1) * (5 - 1) = 2 * 4 = 8$$

Vediamo che anche con questo metodo $\phi(15) = 8$, semplicemente non abbiamo dovuto calcolare tutti i numeri coprimi ad 15, risparmiandoci così molto tempo.

Dalla funzione toziente di Eulero nasce un altro importantissimo teorema: **Teorema di Eulero aritmetica modulare**, che afferma che dati 2 numeri a e n coprimi tra loro, allora possiamo affermare che:

$$a^{\phi(n)} \equiv 1 \mod n \tag{2}$$

Per capirlo meglio prendiamo il solito esempio con $n = 15$. Vuol dire che se prendiamo un qualsiasi numeri coprimo ad 15 (cioè un numero all'interno di Z_{15}^*) abbiamo che la divisione per 15 darà resto sempre 1.

Vediamo con 3 numeri coprimi ad 15: $a = 2$, $a = 7$, $a = 13$.

$$\frac{a^{\phi(n)}}{n} = \frac{2^8}{15} = \frac{256}{15} = 17 \ \textit{resto} \ 1$$

$$\frac{7^8}{15} = \frac{5.764.801}{15} = 384.320 \ \textit{resto} \ 1$$

$$\frac{13^8}{15} = \frac{815.730.721}{15} = 54.382.048 \ \textit{resto} \ 1$$

Mentre questa formula non funziona se a e n non sono coprimi. Per vederlo teniamo $n = 15$ ma usiamo $a = 5$, $a = 9$

$$\frac{5^8}{15} = \frac{390.625}{15} = 26.041 \ \textit{resto} \ 10$$

$$\frac{9^8}{15} = \frac{43.046.721}{15} = 2.869.781 \ \textit{resto} \ 6$$

7.3 L'algoritmo

Dopo aver rivisto tutte le basi neseccarie possiamo vedere come funziona l'algoritmo RSA. Partiamo prendendo 2 numeri primi molto grandi, che chiameremo **p** e **q**. Solitamente questi numeri sono a 2048bit, più è grande il numero più è sicuro l'algoritmo. In sucessione definiamo n come

$$n = p * q$$

In successione calcoliamo $\phi(n)$, e visto che n è composto da solamente 2 numeri primi (**p** e **q** per l'appunto), allora

$$\phi(n) = (p-1) * (q-1)$$

In sucessione scegliamo un numero e che deve essere minore di $\phi(n)$ e coprimo a $\phi(n)$. Spesso viene scelto un numero primo piccolo come 65537 o qualsiasi altro numero primo piccolo, in modo che siamo sicuri che sia coprimo ad $\phi(n)$. In sucessione calcoliamo il numero d che l'**inverso modulare** di e in modulo $\phi(n)$, ovvero quel numero che

$$e * d \equiv 1 \mod \phi(n)$$

Per calcolaro si usa l'**algoritmo di Eulero esteso**, che però in questa dispensa non verrà trattato. A questo punto abbiamo generato le chiavi del nostro algoritmo. La chiave pubblica infatti è data dalla combinazione di (**n, e**), mentre la chiave privata è la combinazione (**n, d**). Per trovare il messaggio cifrato (indicato con C) da un messaggio M (che è la rappresentazione numerica di un messaggio, per esempio interpretando i bit del messaggio testuale come se fossero un numero)

$$C = M^e \mod n$$

$$M = C^d \mod n$$

Un pseudo-codice di un dispositivo che vuole iniziare una comunicazione RSA:

```
p = numeroPrimo()
q = numeroPrimo()
n = p * q
phi = (p-1) * (q-1)
e = 65557
d = inversoModulare(e, phi)

chiavePublica = (n, e)
chiavePrivata = (n, d)
```

Supponiamo di inviare una stringa ad un PC qualsiasi (a cui richiederemo la sua chiave publica),Un possibile pseudo-codice per mandare il messaggio:

```
chiavePublica = richiediChiave()
messaggio = ""
messCriptato = cripta(messaggio, chiavePublica)
inviaMessaggio(messCriptato)
```

Un prossibile script python:

```
n, e = richiediChiave()
messaggio = "Ciao io sono il primo PC"
mess_byte = bytes_to_long(messaggio.encode())
messCriptato = pow(mess_byte, n, e)
print(messCriptato)
```

Mentre il pc che riceverà il messaggio per decriptarlo:

```
messCriptato = riceviMessaggio()
messaggio = decripta(messCriptato, chiavePrivata)
stampa(messaggio)
```

in python:

```
messCriptato = riceviMessaggio()
mess = pow(messCriptato, n, d)
messaggio = long_to_bytes(mess)
print(messaggio)
```

7.4 Come mai funziona?

Per capire come mai funziona questo algoritmo, dobbiamo riprendere l'equazione

$$e * d \equiv 1 \mod \phi(n)$$

E grazie all'equazione n1 sappiamo che possiamo riscriverla nel seguente modo

$$e * d = k * \phi(n) + 1$$

Con k un numero intero positivo, ma non ci interessa il valore. Da questo punto per vedere perchè funziona riprendiamo l'equazione ricordandoci che $C = M^e$

$$C^d = (M^e)^d = M^{e*d} \mod n$$

ora possiamo sistituire l'equazione di prima

$$M^{e*d} = M^{k*\phi(n)+1} = M^{k*\phi(n)} * M^1 = (M^{\phi(n)})^k * M \mod n$$

adesso se vediamo la prima parte pella moltiplicazione vediamo che abbiamo $M^{\phi(n)}$ mod n, che grazie all teorima di Eulero sappiamo, che tutto questo è uguale a 1 in modulo n

$$M^{\phi(n)} \equiv 1 \mod n$$

e quindi tornando all'equazione di prima

$$1^k * M \mod n$$

che semplificando viene fuori che

$$C^d = M \mod n$$

N.B. nel teorema di Eulero chiedeva una restrizione importante, ovvero che M e n siano coprimi tra loro, cosa che ho tralasciato. Questo però non è un problema visto che n è il prodotto di p e q, due numeri primi molto grandi (si parla di numeri anche con 400 cifre), quindi n ha come divisori in comune solo p e q. Pertanto basta che M sia diverso da p e q, ma visto che p e q sono numeri enormi è impossibile che un messaggio sia così tanto grande. Anche lo fosse basta spezzarlo in 2 (o più) messaggi più piccoli.

7.5 Esempio pratico

Per ricapitolare proviamo a vedere un esempio con dei numeri piccoli a 2 cifre, ma ricordiamoci che nella vita reale i numeri sono molto più grandi.

Prendiamo come numeri $p = 61$ e $q = 53$ entrambi numeri primi. Con questo possiamo calcolare n e $\phi(n)$

$$n = p * q = 61 * 53 = 3233$$

$$\phi(3233) = (p-1) * (q-1) = (61-1) * (53-1) = 3120$$

ora scegliamo un numero e minore di 3120 ($\phi(n)$), per comodità scelgo un numero primo piccolo come $e = 17$. Da questo grazie all'algoritmo di Euclide posso calcolare $d = 2753$, perchè

$$17 * 2753 \equiv 1 \mod 3120$$

Ora che abbiamo calcolato tutte le chiavi proviamo a cifrare la lettera **h**, che secondo la tabella ascii ha valore $M = 68$

$$C = M^e = 68^{17} \mod 3233 = 1759$$

Ora la nostra lettera cifrata vale 1759, per decifrare la lettera dobbiamo fare i seguenti calcoli

$$M = C^d = 1759^{2753} \mod 3233 = 68$$

Infatti vediamo che siamo tornati al punto di partenza. Per chi avesse molto tempo libero e volesse esercitarsi in questo argomento provare a decifrare il seguente messaggio:

924641044506842334208913942928831562265038398988867352377330263757334761727

I parametri dell'algoritmo sono:
p = 21866574476608863194802238537987205 4023
q = 18360086601161972010033283472913569 9401
e = 65537

7.6 Crackabilità

Con il termine ***Crackabilità*** intendo la possibilità e la facilità di indovinare la chiave privata di un sistema RSA. Mettiamoci nei panni di un ipotetico hacker che vuole leggere e decriptare i messaggi in una comunicazione RSA. L'hacker in questione avrà a sua disposizione le chiavi publiche dei 2 host, che sono composte da **n** e **e**. Ricordiamo che la chiave privata è composta da **n** e **d**, e che quindi all'hacker manca soltanto **d**.

Infatti un attacco possibile è quello di scoprire quanto vale **d**, per farlo basta vendere come si calcolava. **d** veniva fuori dalla equazione :

$$d * e \equiv 1 \mod \phi(n)$$

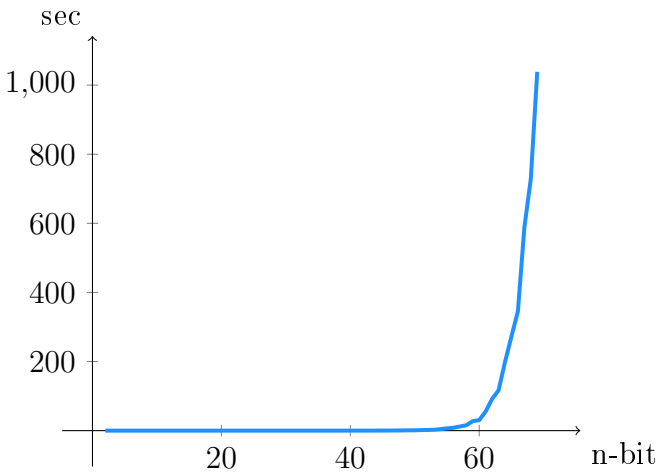
noi però conosciamo **e** perchè è parte della chiave publica, quindi per trovare **d** basta sapere quanto vale $\phi(n)$. Ricordiamo l'equazione da dove viene fuori $\phi(n)$:

$$\phi(n) = (p-1) * (q-1)$$

dove **p** e **q** sono gli unici divisori di **n**. In più **n** noi lo conosciamo perchè fa parte della chiave publica. Quindi per trovate quanto vale **d** basta trovare i divisori di **n** per poi trovare $\phi(n)$ e infine grazie all'algoritmo di Euclide troavre per l'appunto **d**.

Chiaramente se fosse così semplice non si userebbe RSA, infatti abbiamo dato per scontato che sia semplice trovare i divisori di un numero. Ricordiamo che noi con RSA stiamo parlando di numeri molto grandi, ma quando dico molto intento MOLTOOOO. Infatti di solito una buona crittografia RSA è composta da numberi di **2048** bit, che in decimale hanno più o meno 600 cifre. Per scompore numeri così grandi ci vogliono molti secondi, anzi anni.

Per farvelo capire vi ho portato un mio persolane "studio". Tramite python ho provato a scomporre dei numeri composti da n-bit. Tramite una libreria python (**Crypto.Util**) ho generato dei numeri casuali partendo da numeri a 3bit, fino a numeri composti da 69bit.



Possiamo notare come il tempo impiegato cresce in maniera esponenziale al crescere dei bit, che era abbastanza plausibile visto che ogni volta che aggiungiamo un bit moltiplichiamo e che quindi a lungo andate diventa esponensiale. La complessita computazionale di individuare se un numero è primo o meno è di $O(\sqrt{n})$.

Codice usato per ricavare i dati dal grafico

```python
from Crypto.Util import number
import math
import time as t

def isPrime(num):
    up = math.ceil(math.sqrt(num))

    if(num%2 == 0):
        return False

    for i in range(3, up, 2):
        if num%i == 0:
            return False
    return True

f = open("dati.txt", "a")
bits = [x for x in range(2, 70)]
for bit in bits:
    num = number.getPrime(bit)
    start = t.time()
    isPrime(num)
    finish = t.time()
    f.write(f"{bit}\t{finish-start}\n")
```

7.7 Un Futuro Problema

Abbiamo costatato che la sicurezza dell'RSA è dovuta alla difficoltà nel scomporre numeri molto grandi. Però potenzialmente basta avere un pc particolarmente potente e si può rompere l'algoritmo.

Negli ultimi anni si stanno sviluppando **I Computer Quantistici**, che non andremo ad analizzare in questo cotso, ma hanno una potentissima caratteristica. Per via della natura di questi computer, possono performare molte operazioni contemporaneamente, dato che un **qubit** (il bit quantistico) può essere contemporaneamente sia 0 che 1, al contrario di un bit che può essere o solo 1 o solo 0. Due qubit invece rappresentato allo stesso tempo tutte le loro possibili combinazioni (00, 01, 10, 11).

$$|\psi\rangle = c_0|00\rangle + c_1|01\rangle + c_2|10\rangle + c_3|11\rangle$$

Con questa particolarità noi possiamo eseguire la stessa operazioni su più numeri contemporaneamente. Grazie a questa potentissima caratteristica, **Peter Shor** nel 1994 invento **L'algoritmo di Shor**, che permette di fattorizzare un numero con un tempo computazionale **polinomiale** $O(n^3 \log(n))$. Il funzionamento dell'algoritmo è molto complesso ma sfrutta sempre i teoremi che abbiamo visto, specialmente quello di Fermat

$$g^r \equiv 1 \mod N$$

E l'algoritmo si basa fondalmentalmente su indovinare l'esponente (**r**) e avicinandosi sempre di più al numero effettivo.

Ad ogni modo questo algoritmo è molto efficiente, ma per fortuna non è ancora utilizzabile perchè non esiste ancora un computer quantistico potente abbastanza. Infatti il numero più grande che un computer quantistico è riuscito a fattorizzare grazie all'algoritmo di Shor è **21**. Nonostante non sia ancora un problema questo algoritmo, per evitare problemi in futuro si stanno sviluppando algoritmi che siano difficili da decriptare anche per dei computer quantistici.

7.7.1 Funzione Zeta Riemann

Un'altro possibile rischio per la sicurezza dell'Algoritmo RSA è la possibilità che si possa trovare una funzione matematica che riesca a generare tutti i numeri primi. In caso esistesse, renderebbe molto più veloce crackare una chiave RSA, passando da un tempo computazionale esponenziale a lineare $O(n)$, anche più veloce di un computer quantistico. Per fortuna non si è ancora trovato una funziona che permetta tale funzionamento, ma una parte della comunità scientifica sta studiando perchè si suppone che la **Funzione Zeta di Riemann** ($\zeta(s)$) possa prevedere i numeri primi.

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$$

Per ora non c'è alcuna dimostrazione a riguardo ma le ricerche sono ancora in atto.

7.8 Principali attacchi RSA

RSA è un algoritmo sicuro e non cracckabile per i computer d'oggi. Però se non viene usato come si deve permette tramite certe tecniche di rompere la crittografia. Ora vediamo tre principali attacchi che si possono effettuare su una crittografia RSA non usata in maniera appropriata.

7.8.1 Low public Exponent

Inizialmente abbiamo detto che l'esponente publico (**e**) deve essere primo, affinche esista un inverso modulare (**d**), quindi può anche essere un numero piccolo come **3**, **7**, **11**. Purtroppo però se usiamo un **e** molto piccolo possiamo inbattere in un problema. Infatti riprendiamo le formule fondamentali di RSA:

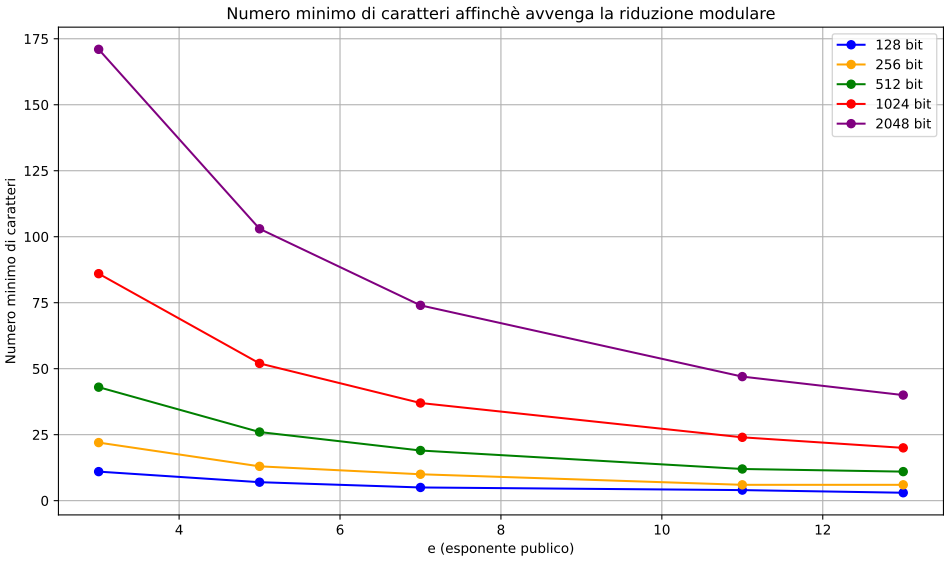
$$C \equiv M^e \mod n$$

Se $M^e < n$ allora non verrà mai fatta la riduzione modulare, perchè il numero non supera il modulo. Con questo il messaccio cifrato sarà soltanto M^e e quindi basta invertire la formula per trovare il messaggio in chiaro

$$C = M^e$$

$$M = \sqrt[e]{C}$$

Verosimilmente anche scegliendo messaggi molto corti (come i singoli caratteri) può essere rischioso. A questo proposito ho fatto un piccolo studio in merito. Ho analizzato quanti caratteri minimi servono per un messaggio per non riscontrare in questo problema, in base alla grandezza di chiavi (ho scelto 4 valori: chiavi a **128bit**, **256bit**, **512bit**,**1024bit** e **2048bit**) e in base a un esponente molto piccolo. In altre parole, ho trovato quanti caratteri servono per un messaggio per avere una riduzione e quindi per essere sicuro, perchè altrimenti basta fare la radice e-esima per tornare al messaggio in chiaro originale.



Notiamo che per valori molto bassi servono molti caratteri affinchè avvenga la riduzione modulare, e soppratutto più è grande la chiave più caratteri servono. Chiaramente per dei valori normali di **e**, come 65337, la riduzione avviene anche con un singolo carattere quindi non è un problema, ma se venisse scelto un esponente piccolo abbiamo visto cosa questo comporta.

7.8.2 P e Q vicini

Un altro problema che può accadere è qualora **p** e **q** fossero valori molto simili. Per capire perchè può essere un problema analiziamo dal punto di vista matematico. Supponiamo che quindi **p** e **q** sono molto simili, e che quindi uno dei due numeri è più grande (supponiamo che q sia il più grande, ma chiaramente è indifferente chi sia il più grande).

$$q = p + \Delta x$$

Indichiamo con Δx la differenza tra i due numeri. Ora proviamo a calcolare il modulo **n**

$$n = p * q = p * (p + \Delta x) = p^2 + p\Delta x$$

Ora analiziamo cosa abbiamo scoperto, perchè ricordiamo che **p** e **q** sono numeri enormi (inorno alle 300/400 cifre) mentre Δx è un numero relativamente piccolo (vedremo dopo quanto piccolo) ma comunque insignificante rispetto a p^2 che ha approssimativamente 600/700 cifre. Quindi il termine con il Δx lo possiamo anche trascurare

$$p^2 + p\Delta x \approx p^2$$

Da questo troviamo che

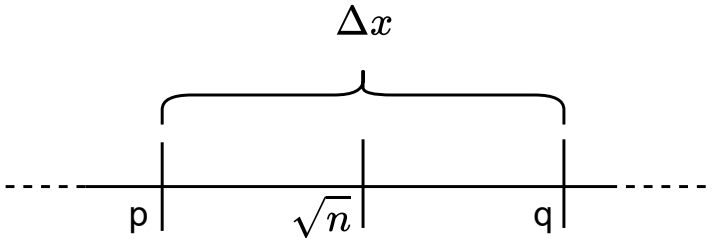
$$n = p^2 + p\Delta x \approx p^2$$

$$n \approx p^2$$

E quindi troviamo una approssimazione molto comoda

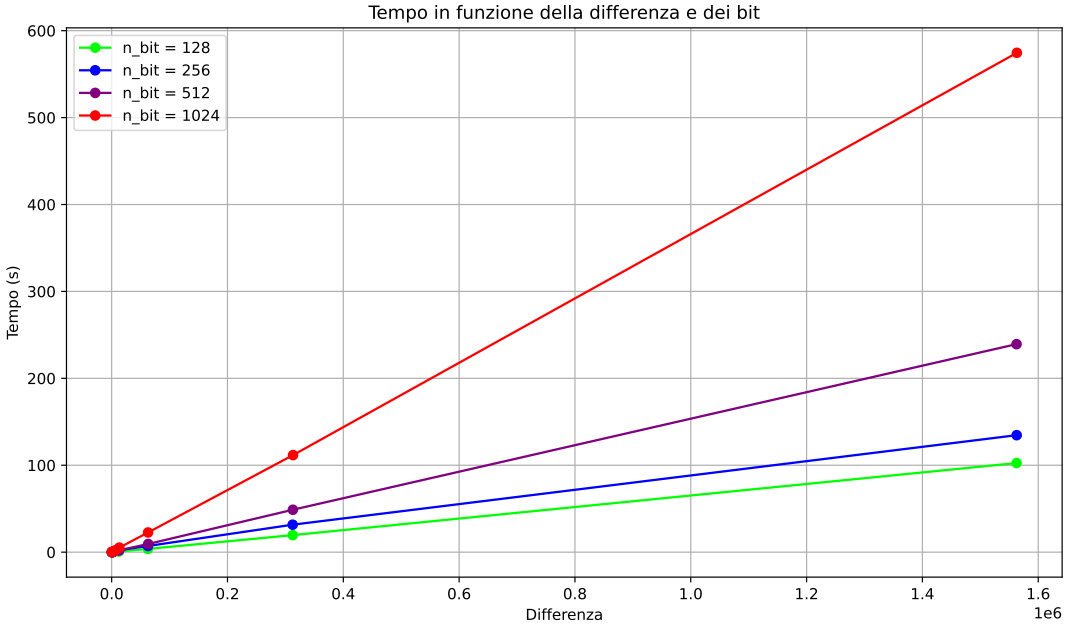
$$p \approx \sqrt{n}$$

Così abbiamo scoperto che la **radice di n** è un numero molto vicino a **p**, perciò si possono cercare tutti i numeri primi vicini a $\sqrt{n}$ per trovare p



Chiaramente più è grande la differenza (Δx) più sarà difficile e, nei casi reali praticamente impossibile usare questa tecnica. Questa tecnica ha una complessita computazionale $o(\frac{\Delta x}{4})$, quindi linearmente proporzionale.

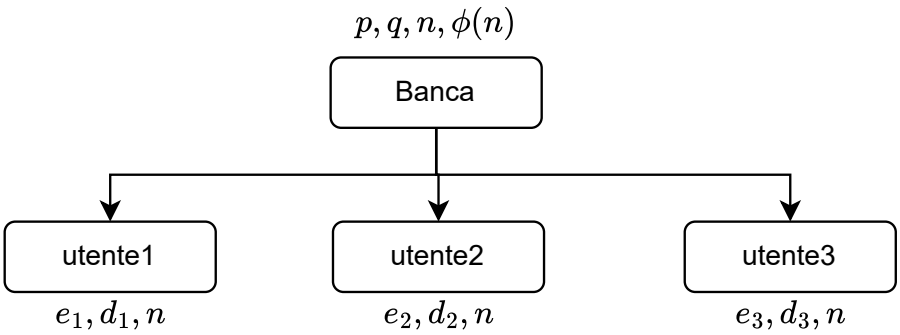
Per capire bene anche questo attacco ho provato a vedere quanto tempo ci si impiega a fattorizzare un numero grande (**n**) con questa tecnica, e ho trovato questo grafico



Notiamo che già con una differenza di 1.000.000 tra i due fattori primi ci si impiega dai 2 minuti a poco meno di 10 minuti. Una differenza di 1.000.000 è veramente poco rispetto ai numeri da 300 cifre, quindi è facile capire che questo attacco non ha una vera e propria implementazione pratica.

7.8.3 Stesso N, diversi esponenti

Supponiamo che una banca debba creare tutte le chiavi per i suoi cliente, visto che la generazione delle chiavi è un processo molto lungo e dispendioso, supponiamo che decida di usare lo stesso modulo (**n**) e quindi stessi fattori primi (**p**, **q**) ma per ogni cliente usare un esponente diverso (**e** e quindi di conseguenza **d**). Inizialmente sembra un'idea molto furba perchè con lo stesso modulo (e quindi evitando tutto il processo di generazione di numeri primi enormi) possiamo dando a ciascun cliente un esponente pubblico diverso (chiaramente deve essere un numero primo).



Vediamo come però come è molto rischioso, perchè ognuno ha la sua coppia di chiavi (**e**, **n**) e (**d**, **n**) ma nessuno conosce i fattori primi, perchè li possiede solo la banca. Quindi con l'attacco che ora vedremo, possiamo **fattorizzarte n** partendo da **e**, **d**. Questo perchè se qualcuno riuscisse a scomporre i fattori primi potrebbe trovare $\phi(n)$ e da li trovare tutti gli esponenti privati di tutti, permettendo così di avere tutte le chiavi del sistema.

Per fattorizzare

8 Diffie-Hellman

9 Elliptic Curve Cryptografy (ECC)