Attacchi al cifrario RSA

Michael De Angelis

A.A 2019-2020

1 Introduzione

L'algoritmo di cifratura **RSA**, basato sull'esistenza di due chiavi distinte, utilizzate per la cifratura e la decifrazione (*algoritmo asimmetrico*), è uno dei cifrari più impiegati in diversi sistemi hardware e software e, fino ad oggi, si è dimostrato sostanzialmente inviolabile.

Nella seguente trattazione daremo una definizione formale dell'organizzazione dell'algoritmo per poi analizzare alcuni attacchi al cifrario volti alla ricerca della **chiave privata** di un utente, in modo da decifrare qualsiasi messaggio cifrato con la sua **chiave pubblica**, oppure volti alla semplice decifrazione di un dato messaggio sfruttando proprietà dell'algebra modulare. Nello specifico andremo a confrontare un attacco a forza bruta, basato sul verificare tutte le soluzioni teoricamente possibili fino a trovare quella effettivamente corretta, con i seguenti attacchi:

- Metodo di fattorizzazione di Fermat;
- Attacco basato sulla scelta dello stesso esponente;
- Attacco basato sulla scelta dello stesso valore di n.

A fini accademici, in modo da eseguire gli attacchi elencati in tempi ragionevoli, sono state limitate le dimensioni delle chiavi utilizzate a 64 bit; si tenga presente che in usi reali le chiavi possono avere lunghezza superiore a 2048 bit!

2 Descrizione del cifrario

Ogni utente che vuole utilizzare il meccanismo di cifratura RSA come prima cosa deve scegliere due numeri primi molto grandi p e q. Abbiamo limitato la dimensione di tali valori a 32 bit e utilizzato i **test di primalità di Miller-Rabin** e **Lucas-Lehmer** per determinare, con buona probabilità (la probabilità che siano composti non è superiore a 2^{-100}), i suddetti valori primi. Per fare ciò è stata utilizzata la funzione *probablePrime* della classe *BigInteger* del linguaggio di programmazione Java.

Una volta scelti tali numeri primi, si calcola $n = p \times q$ e la funzione di Eulero $\phi(n) = (p-1)\times(q-1)$. Possiamo osservare come sia n che $\phi(n)$ saranno variabili a 64 bit.

A questo punto si sceglie un valore intero e minore di $\phi(n)$ e primo con questo. Nell'implementazione adottata e è una variabile a 32 bit.

Infine, si calcola l'intero $d = e^{-1} \mod \phi(n)$; l'esistenza e l'unicità di d è assicurata dal fatto che e e $\phi(n)$ sono primi tra loro. Il calcolo di quest'ultimo passaggio è stato effettuato sfruttando la funzione *modInverse*; tale funzione fa uso dell'algoritmo di **Euclide Esteso**. d sarà una variabile a 64 bit.

Si considerano quindi la chiave pubblica, formata dalla coppia (e, n), e la chiave privata (d). [1]

2.1 Cifratura e Decifrazione

Dato un messaggio m codificato in binario, trattato come numero intero, tale che esso sia minore di n (in caso contrario è sempre possibile suddividere il messaggio in blocchi di al più $\lfloor \log_2 n \rfloor$ bit), la funzione di cifratura utilizzata è $c = m^e \mod n$ mentre quella di decifrazione sarà $m = c^d \mod n$.

2.2 Correttezza di RSA

Theorem 1. *Per qualunque intero $m < n$ si ha $(m^e \mod n)^d \mod n = m$, dove n, e e d sono parametri del cifrario RSA. [1]*

Proof. La relazione da dimostrare può essere scritta come $m^{ed} \mod n = m$. Distinguiamo due casi, relativi ai valori p e q scelti dall'utente destinatario e al valore di m scelto dal mittente:

- *p e q non dividono m*: Si ha $mcd(m, n) = 1$ quindi per il teorema di Eulero risulta che $m^{\phi(n)} \equiv 1 \mod n$. Poiché d è l'inverso di $e \mod \phi(n)$ abbiamo $e \times d \equiv 1 \mod \phi(n)$, ovvero $e \times d = 1 + r\phi(n)$ con r intero positivo. Otteniamo quindi $m^{ed} \mod n = m^{1+r\phi(n)} \mod n = m \times (m^{\phi(n)})^r \mod n = m \times 1^r \mod n = m$;
- *p (oppure q) divide m, ma q(oppure p non divide m*: Poiché p divide m abbiamo $m \equiv m^r \equiv 0 \mod p$, cioè $(m^r - m) \equiv 0 \mod p$, per qualunque intero positivo r. Similmente al punto precedente abbiamo $m^{ed} \equiv m \mod q$ e quindi $(m^{ed} - m) \equiv 0 \mod q$. A questo punto basta notare che $m^{ed} - m$ è divisibile sia per p che per q e di conseguenza lo è per il loro prodotto n;
- *p e q dividono entrambi m*: Questa situazione non può accadere perché si avrebbe $m >= n$ che va contro l'ipotesi sulla dimensione dei blocchi.

□

2.3 Test di Primalità di Miller-Rabin e Lucas-Lehmer

Come precedentemente accennato, la funzione della classe BigInteger che ci permette di generare in modo casuale, probabilistico e efficiente i valori p e q si basa su questi due risultati teorici.

Definition 2.1. Test di Miller-Rabin Sia n un numero intero positivo dispari e non primo. I numeri positivi $b < n$ tali che $mcd(b, n) = 1$ e tali che n sia uno *pseudoprimo di Eulero forte* in base b sono non più di un quarto di tutti i numeri positivi $b < n$ tali che $mcd(b, n) = 1$. [9]

Lemma 2. Pseudoprimo di Eulero Un numero n è detto pseudoprimo di Eulero in base a (con $mcd(n, a) = 1$)) se è un numero dispari composto e $a^{(n-1)/2} \equiv \pm 1 \mod n$. [6]

La complessità computazionale è polinomiale mentre per l'affidabilità sappiamo che la probabilità che n non sia primo e sia uno pseudoprimo forte in base b_m è minore di $1/4$, quindi, la probabilità che n non sia primo è minore di $1/4^m$.

Definition 2.2. Test di Lucas-Lehmer Sia p un numero primo. Il corrispondente *numero di Mersenne* $M_p = 2^p - 1$ è primo se e solo se $L_{p-1} \equiv 0 \mod M_p$ dove L_n è l'ennesimo termine della successione definita ricorsivamente come $L_{n+1} = {L^2}_n - 2$ a partire da $L_1 = 4$. [8]

Lemma 3. Primo di Mersenne Un numero primo di Mersenne è un numero primo inferiore di uno rispetto ad una potenza di due; è quindi esprimibile come $M_p = 2^p - 1$ con p intero positivo primo. [5]

Così come per Miller-Rabin il test ha complessità polinomiale.

2.4 Algoritmo di Euclide Esteso

Come detto poc'anzi, la funzione della classe BigInteger (e un'implementazione personale, fornita nella bibliografia, scritta per permettere di recuperare valori che la funzione suddetta non restituisce) che permette il calcolo dell'inverso moltiplicativo, utilizza il seguente risultato teorico:

Definition 2.3. Algoritmo di Euclide Esteso L'algoritmo di Euclide Esteso è un'estensione dell'algoritmo di Euclide che permette di calcolare non solo il massimo comun divisore tra due interi a e b ma anche i coefficienti dell'identità di Bézout x e y tali che $ax + by = mcd(a, b)$. L'algoritmo è particolarmente utile quando a e b sono interi coprimi: in questo caso x è l'inverso moltiplicativo di $a \mod b$ e y è l'inverso moltiplicativo di $b \mod a$. [2]

Lemma 4. Identità di Bézout L'identità di Bézout afferma che se a e b sono interi(non entrambi nulli) e il loro massimo comun divisore è d, allora esistono due interi x e y tali che $ax + by = d$. [3]

3 Attacco a Forza Bruta

I metodi a forza bruta verificano tutte le soluzioni teoricamente possibili fino a trovare quella corretta; se da una parte questi metodi consentono, teoricamente, di trovare sempre la soluzione corretta, dall'altra è sempre la soluzione più lenta e/o dispendiosa. Nella seguente trattazione utilizzeremo un metodo a forza bruta per fattorizzare la componente n della chiave pubblica e ricavare quindi i numeri primi p e q in modo da calcolare la corrispondete chiave privata d, essendo a conoscenza della componente e della chiave pubblica. [4]

Per la decifrazione di un crittogramma c sarebbe sufficiente calcolare la sua radice e-esima modulo n, essendo a conoscenza della chiave pubblica (e, n) e che $c = m^e \mod n$; però, il calcolo della radice e-esima nell'algebra modulare può essere eseguito efficientemente se il modulo n è primo, ma è difficile quanto la fattorizzazione se n è composto. Si ha quindi che il calcolo di m a partire da

e, n e c non è più facile del calcolo dei fattori primi di n; è quindi più conveniente eseguire la fattorizzazione di n in quanto questo comporta la forzatura del cifrario e non la singola decifrazione del particolare crittogramma. [1]

La fattorizzazione della componente n della chiave pubblica avviene quindi effettuando il calcolo $n \mod i$, per i da 1 a $2^{32} - 1$, fino a quando non si trova un valore i per il quale il risultato sia zero. Trovato tale valore di i, questo sarà uno dei due fattori di n; derivare quindi il secondo fattore è immediato dividendo n per i. Infine, avendo a disposizione e si calcola $e^{-1} \mod n$ per ricavare la chiave privata: il cifrario è quindi violato.

3.1 Simulazione

La simulazione di un attacco a forza bruta avviene istanziando un oggetto RSA utilizzando la corrispondente classe, costruita dotandola dei metodi necessari alla creazione di tutti i parametri necessari ad effettuare la cifratura dei messaggi e la decifrazione dei crittogrammi (seguendo le regole e le condizioni esposte), e passando alla funzione *bruteForceAttack* della classe *RsaAttacker* (anch'essa definita in modo da contenere tutti i metodi corrispondenti ai vari attacchi) i valori della chiave pubblica dell'oggetto RSA stesso. Eseguito l'attacco verrà stampato il tempo impiegato per violare il cifrario.

4 Metodo di fattorizzazione di Fermat

Una delle regole fondamentali da tenere in considerazione nella scelta dei primi p e q, del cifrario, consiste nel fatto che la differenza tra questi due deve essere grande. Avendo scelto di utilizzare valori a 32 Bit per le componenti p e q del cifrario, tali valori saranno necessariamente vicini tra loro. Se il numero $|p - q|$ fosse piccolo, $(p + q)/2$ sarebbe prossimo a $\sqrt{n}$; questa informazione consentirebbe di determinare velocemente p e q. Sappiamo che $\frac{(p+q)^2}{4} - n = \frac{(p-q)^2}{4}$, che mostra come il membro sinistro sia un quadrato perfetto. Poiché $(p + q)/2 > \sqrt{n}$, si possono scandire gli interi maggiori di $\sqrt{n}$ fino a trovare un intero z tale che $z^2 - n = w^2$ sia un quadrato perfetto; a questo punto si possono inferire i valori $p = z + w$ e $q = z - w$. Di conseguenza, avendo a disposizione p, q, n ed e si può calcolare la chiave privata: il cifrario è quindi violato. [1]

4.1 Simulazione

La simulazione di un attacco tramite la Fattorizzazione di Fermat avviene istanziando un oggetto RSA , come nella simulazione di un attacco a forza bruta, e passando alla funzione *fermatFactoringAttack* (sempre della classe RsaAttacker) i valori della chiave pubblica dell'oggetto stesso. Analogamente alla simulazione dell'attacco a forza bruta, terminato l'attacco verrà stampato il tempo impiegato per violare in cifrario.

5 Attacco sulla scelta dello stesso esponente

La scelta del valore e riveste un ruolo importante; per prima cosa deve valere $e \neq (\phi(n) + 2)/2$ in quanto tale valore non indurrebbe alcuna trasformazione del messaggio in chiaro. Inoltre, si consiglia di scegliere per e un valore piccolo (solitamente 65537; nella seguente trattazione viene generato casualmente) per velocizzare il processo di cifratura. Poniamo che e sia molto piccolo e che molti utenti abbiano scelto lo stesso valore, consideriamo inoltre che almeno e di questi utenti ricevano il medesimo messaggio m. Si ha quindi che: $c_1 = m^e \mod n_1, c_2 = m^e \mod n_2, ..., c_e = m^e \mod n_e$, dove $m < n_i$, $1 \leq i \leq e$, per la generazione di $c_1, ..., c_e$. Possiamo assumere che $n_1, n_2, ..., n_e$ siano primi tra loro poiché la probabilità che ciò non avvenga è trascurabile, e in caso contrario un crittoanalista potrebbe fattorizzarli calcolandone, a coppie, il massimo comun divisore. Per il *Teorema cinese del resto* esiste un unico $m' < n = n_1 \times n_2 \times ... \times n_e$ che soddisfa l'equazione $m' \equiv m^e \mod n$. Poiché $m < n_i$ per ogni i, abbiamo $m^e < n$: possiamo quindi calcolare il valore di m' attraverso il teorema cinese del resto e da questo calcolare m mediante l'estrazione della radice e-esima (eseguibile in modo efficiente perché non interviene l'operazione in modulo). [1]

Definition 5.1. Teorema cinese del resto Si supponga che $n_1, n_2, ..., n_k$ siano interi a due a due coprimi (il che significa che $mcd(n_i, n_j) = 1$ quando $i \neq j$). Allora, comunque si scelgano degli interi $a_1, a_2, ..., a_k$, esiste un intero x soluzione del sistema di congruenze $x \equiv a_i \mod n_i$ per $i = 1, ..., k$. Inoltre, tutte le soluzioni x di questo sistema sono congruenti modulo il prodotto $n = n_1 \times n_2 \times ... \times n_k$. [7]

5.1 Simulazione

Per la simulazione dell'attacco basato sullo stesso valore di e, si creano e istanze della classe RSA (per la simulazione si è posto $e = 3$), rappresentanti degli e utenti, con lo stesso valore e della chiave pubblica e valori di n differenti. Dati gli e valori di n degli e utenti, $n_1, n_2, ..., n_e$, si crea in modo casuale il messaggio m in modo che sia minore del più piccolo degli e valori di n. Per ogni oggetto RSA si cifra il messaggio m in modo da ottenere $c_1, c_2, ..., c_e$. A questo punto si passano alla funzione *eSameValueAttack* il valore di e, la lista $n_1, n_2, ..., n_e$ e la lista $c_1, c_2, ..., c_e$. Terminato l'attacco verrà stampato il messaggio ricavato dall'esecuzione della violazione e il tempo richiesto: il messaggio inviato agli e utenti è così scoperto.

Il teorema cinese del resto è stato implementato attraverso un approccio iterativo in cui in ogni passo, per $0 \leq i \leq e$, si esegue $p = n/n_i$ da cui si ricava $tmp = p^{-1} \mod n_i$; infine si somma ad una variabile *sum* (inizializzata a zero) $c_i \times tmp \times p$. Terminato il ciclo si effettua l'estrazione della radice e-esima.

6 Attacco sulla scelta dello stesso valore di n

Così come la scelta del valore e, anche la scelta di n deve essere ben oculata; è necessario infatti che utenti diversi scelgano valori di n diversi fra loro, anche se poi le chiavi potrebbero essere differenziate mediante la scelta di e. Se questa eventualità dovesse accadere, un crittoanalista potrebbe selezionare due utenti con chiavi pubbliche (e_1, n) e (e_2, n) tale che $mcd(e_1, e_2) = 1$, il che è molto probabile se e_1 e e_2 sono scelti in modo casuale. In tal caso esistono due interi r e s calcolabili con l'algoritmo di Euclide esteso per cui $e_1 r + e_2 s = 1$. Si consideri il caso in cui $r < 0$, il caso $s < 0$ è simmetrico, e che il crittoanalista riesca a intercettare due crittogrammi c_1 e c_2 relativi al medesimo messaggio m. Si ha quindi che $m = m^{e_1 r + e_2 s} = (c_1^r \times c_2^s) \mod n = ((c_1^{-1})^{-r} \times c_2^s) \mod n$. Poiché c_1 e n sono primi tra loro (se non lo fossero risulterebbe $c_1 = p$ o $c_1 = q$, ma è molto improbabile), il crittoanalista può a questo punto calcolare $c_1^{-1} \mod n$; e quindi calcolare $(c_1^{-1})^{-r}$ poiché $-r$ è positivo. Infine, si calcola c_2^s e quindi possiamo ricostruire m attraverso l'espressione di cui sopra. [1]

6.1 Simulazione

Per la simulazione dell'attacco basato sullo stesso valore di n, si creano due istanze della classe RSA, rappresentanti i due utenti, con lo stesso valore n della chiave pubblica e con esponenti tali che $mcd(e_1, e_2) = 1$. Si crea quindi in modo casuale un messaggio m in modo che sia minore di n. A questo punto si passano alla funzione *nSameValueAttack* il valore di n, gli esponenti delle due chiavi pubbliche e il messaggio cifrato dai due utenti (c_1 e c_2). Terminato l'attacco verrà stampato il messaggio ricavato dall'esecuzione della violazione e il tempo richiesto: il messaggio inviato ai due utenti è così scoperto.

7 Comparazione degli attacchi

Per effettuare il Testing degli attacchi descritti ed avere al contempo una buona approssimazione del tempo richiesto per l'esecuzione di questi ultimi, dato che, fatta eccezione per l'attacco a forza bruta, il tempo d'esecuzione dei restanti approcci è generalmente molto basso è stato deciso di considerare una simulazione di questi ultimi come una successione di simulazioni, definite esattamente come trattato poc'anzi, e considerando come tempo d'esecuzione totale la somma dei tempi necessari ad eseguire ogni singola simulazione per l'attacco considerato.

7.1 Testing Attacco a Forza Bruta

Come detto, per la simulazione dell'attacco a forza bruta, avendo generalmente tempi di esecuzione elevati, una simulazione di tale attacco corrisponde ad una singola esecuzione del processo descritto nella sezione 3.1. Eseguiamo quindi un set di 10 simulazioni e calcoliamo da esso il tempo medio di esecuzione dell'attacco a forza bruta su chiavi di 64 Bit.

Table 1: Tempo richiesto - [ms] - Forza Bruta

Case	Forza Bruta
1	130990
2	194610
3	143078
4	166019
5	115731
6	130572
7	181297
8	125023
9	121657
10	157119

Il tempo medio richiesto da un attacco a Forza Bruta è quindi di $146609,6ms$.

7.2 Testing Metodo di Fattorizzazione di Fermat

Relativamente a quanto anticipato, la simulazione del Metodo di Fattorizzazione di Fermat, in un primo set di 10 simulazioni, corrisponde all'esecuzione del processo descritto nella sezione 4.1 10 volte per ogni iterazione (per un totale di 100 prove), in modo da considerare l'errore totale come la somma del tempo necessario all'esecuzione di ogni singola prova; in un secondo set di 10 simulazioni le prove aumenteranno a 20 (per un totale di 200) e a 30 nel terzo ed ultimo set (300 prove complessive). Osserviamo quindi i tempi di esecuzione per la realizzazione di 10 prove per ognuna delle 10 simulazioni:

Table 2: Tempo richiesto - [ms] - Fattorizzazione di Fermat - 20 prove

Case	Fattorizzazione di Fermat	Media singola prova
1	30837	3083,7
2	93240	9324
3	27980	2798
4	44353	4435,3
5	24678	2467,8
6	40154	4015,4
7	13720	1372
8	37011	3701,1
9	34279	3427,9
10	61612	6161,2

Nel primo set di simulazione, il tempo di esecuzione medio per l'esecuzione di 10 attacchi basati sul Metodo di Fattorizzazione di Fermat è $40786,4ms$, per una media di $4078,64ms$ per una singola prova di una simulazione.

Per il set di simulazioni caratterizzato dall'esecuzione di 20 prove per ogni iterazione, si ha:

Table 3: Tempo richiesto - [ms] - Fattorizzazione di Fermat - 20 prove

Case	Fattorizzazione di Fermat	Media singola prova
1	71412	3570,6
2	69298	3464,9
3	105585	5279,25
4	53070	2653,5
5	101852	5092,6
6	111878	5593,9
7	107318	5365,9
8	54407	2720,35
9	96754	4837,7
10	86156	4307,8

Nel secondo set di simulazione, il tempo di esecuzione medio per l'esecuzione di 20 attacchi basati sul Metodo di Fattorizzazione di Fermat è $85773ms$, per una media di $4288,65ms$ per una singola prova di una simulazione.

Per quanto concerne invece il set caratterizzato dall'esecuzione di 30 prove per ogni iterazione, abbiamo:

Table 4: Tempo richiesto - [ms] - Fattorizzazione di Fermat - 30 prove

Case	Fattorizzazione di Fermat	Media singola prova
1	168199	5606,63
2	132142	4404,73
3	96701	3223,36
4	191799	6393,3
5	178794	5959,8
6	121116	4037,2
7	126634	4221,13
8	128191	4273,03
9	166277	5542,56
10	134901	4496,7

Nel terzo set di simulazione, il tempo di esecuzione medio per l'esecuzione di 30 attacchi basati sul Metodo di Fattorizzazione di Fermat è $144475,4ms$, per una media di $4815,84ms$ per una singola prova di una simulazione.

7.3 Testing Attacco sulla scelta dello stesso esponente

Esattamente come proposto per il Metodo di Fattorizzazione di Fermat, una singola simulazione di un attacco basato sulla scelta dello stesso esponente consiste

nel reiterare il processo descritto nella sezione 5.1. Nel primo set di 10 simulazioni, il numero di prove per ogni iterazione è fissato a 5000, per un totale di 50000; nuovamente, come fatto per il metodo precedente, le prove saranno aumentate a 10000 e 20000 nel secondo e nel terzo per un totale, rispettivamente, di 100000 e 200000 prove. Osserviamo quindi i tempi di esecuzione per la realizzazione di 5000 prove per ognuna delle 10 simulazioni:

Table 5: Tempo richiesto - [ms] - Stesso Esponente - 5000 prove

Case	Stesso Esponente	Media singola prova
1	15676	3,13
2	15533	3,10
3	15118	3,02
4	15854	3,17
5	14911	2,98
6	14746	2,94
7	15229	3,04
8	15008	3,00
9	16336	3,26
10	15246	3,04

Nel primo set di simulazioni, il tempo di esecuzione medio per l'esecuzione di 5000 attacchi basati sullo stesso valore dell'esponente è $15365,7ms$, per una media di $3,07ms$ per una singola prova di una simulazione.

Per il set di simulazioni composto dall'esecuzione di 10000 prove per ognuna di esse, abbiamo:

Table 6: Tempo richiesto - [ms] - Stesso Esponente - 10000 prove

Case	Stesso Esponente	Media singola prova
1	30103	3,01
2	30603	3,06
3	31172	3,11
4	34735	3,47
5	30272	3,02
6	33029	3,30
7	29403	2,94
8	29141	2,91
9	30051	3,00
10	29800	2,98

Nel secondo set di simulazione, il tempo di esecuzione medio per l'esecuzione di 10000 attacchi basati sullo stesso valore dello stesso esponente è $30830,9ms$, per una media di $3,08ms$ per una singola prova di una simulazione.

Infine, per il set di simulazioni caratterizzato dall'esecuzione di 20000 prove per ogni simulazione:

Table 7: Tempo richiesto - [ms] - Stesso Esponente - 20000 prove

Case	Stesso Esponente	Media singola prova
1	59271	2,96
2	58692	2,93
3	58817	2,94
4	65114	3,25
5	62102	3,10
6	58390	2,91
7	60699	3,03
8	62006	3,10
9	57793	2,88
10	58345	2,91

Nel terzo set di simulazione, il tempo di esecuzione medio per l'esecuzione di 20000 attacchi basati sullo stesso valore dello stesso esponente è $60122,9ms$, per una media di $3,00ms$ per una singola prova di una simulazione.

7.4 Testing Attacco sulla scelta dello stesso n

Analizziamo quindi l'ultimo attacco descritto applicando il medesimo modus operandi che abbiamo adottato per i precedenti test. Il numero di prove eseguite per ogni simulazione nel primo set viene fissato a 15000 per un totale di 150000; il secondo set è caratterizzato da 30000 prove per ciascuna simulazione (300000 in totale); nell'ultimo set invece si conducono 60000 prove a simulazione per un totale di 600000 prove. Di seguito osserviamo i tempi di esecuzione per la realizzazione di 15000 prove per ognuna delle 10 simulazioni del primo set:

Table 8: Tempo richiesto - [ms] - Stesso n - 15000 prove

Case	Stesso n	Media singola prova
1	14444	0,96
2	13829	0,92
3	14092	0,93
4	13788	0,91
5	13033	0,86
6	13587	0,90
7	14662	0,97
8	13593	0,90
9	13986	0,93
10	13561	0,90

Nel primo set di simulazioni, il tempo di esecuzione medio per l'esecuzione di

5000 attacchi basati sullo stesso valore di n è $13857,5ms$, per una media di $0,92ms$ per una singola prova di una simulazione.

Nel secondo set, composto da 30000 prove per ognuna delle 10 simulazioni, riscontriamo la seguente situazione:

Table 9: Tempo richiesto - [ms] - Stesso n - 30000 prove

Case	Stesso n	Media singola prova
1	26678	0,88
2	27547	0,91
3	27156	0,90
4	26186	0,87
5	26853	0,89
6	26291	0,87
7	27588	0,91
8	26172	0,87
9	27114	0,90
10	23919	0,79

Nel secondo set di simulazione, il tempo di esecuzione medio per l'esecuzione di 30000 attacchi basati sullo stesso valore di n è $26550,4ms$, per una media di $0,88ms$ per una singola prova di una simulazione.

In ultima istanza, il comportamento sull'ultimo set, definito dall'esecuzione di 60000 prove per simulazione:

Table 10: Tempo richiesto - [ms] - Stesso n - 60000 prove

Case	Stesso n	Media singola prova
1	52216	0,87
2	51332	0,85
3	52873	0,88
4	51501	0,85
5	51614	0,86
6	51905	0,86
7	49995	0,83
8	52092	0,86
9	51012	0,85
10	51831	0,86

Nel terzo set di simulazione, il tempo di esecuzione medio per l'esecuzione di 60000 attacchi basati sullo stesso valore di n è $51637,1,4ms$, per una media di $0,86ms$ per una singola prova di una simulazione.

8 Conclusioni

Possiamo quindi osservare come l'attacco a forza bruta sia molto più inefficiente rispetto ad attacchi più informati; inoltre, se le condizioni per eseguirli sono soddisfatte, questi ultimi possono essere applicati molteplici volte su chiavi pubbliche / messaggi differenti nello stesso arco di tempo necessario all'esecuzione dell'attacco a forza bruta.

Si tenga inoltre in considerazione che tali risultati sono, come precedentemente riportato, il risultato della simulazione degli attacchi su parametri del cifrario RSA di dimensioni considerevolmente più ristrette rispetto a quelle utilizzate in applicazioni reali. Se la dimensione delle chiavi fosse di migliaia di bit attacchi di tipo forza bruta sarebbero esponenzialmente più dispendiosi e inapplicabili. Per ovviare attacchi basati sulla fattorizzazione di Fermat, usando p e q molto più grandi, è facile trovare valori per i quali la loro differenza in valore assoluto non è piccola. Gli attacchi basati sugli stessi valori dell'esponente o di n invece devono essere ovviati, ad esempio all'interno di una stessa organizzazione, semplicemente con delle scelte oculate di questi ultimi.

A Codice - RSA

```java
/**
 * ########################################################
 *
 * @author: Michael De Angelis
 * @mat: 560049
 * @project: Esperienze di Programmazione [ESP]
 * @AA: 2019 / 2020
 *
 * ########################################################
 */

package rsa_attacks;

import java.util.Random;
import java.math.BigInteger;

public class RSA {

   private BigInteger p32Bit;
   private BigInteger q32Bit;
   private BigInteger n64Bit;
   private BigInteger phiN64Bit;
   private BigInteger e32Bit;
   private BigInteger privateKey;

   private void initializer(BigInteger p32Bit, BigInteger q32Bit) {
      // Compute n = p * q
      n64Bit = p32Bit.multiply(q32Bit);

      // Compute phiN = (p - 1) * (q - 1)
      phiN64Bit = p32Bit.subtract(BigInteger.ONE)
      .multiply(q32Bit.subtract(BigInteger.ONE));

      // Compute e
      e32Bit = eSelection(phiN64Bit);

      // Compute d = e ^ (-1) mod phiN
      privateKey = e32Bit.modInverse(phiN64Bit);
   }

   // Private method to find the exponent e
   private static BigInteger eSelection(BigInteger phiN64Bit) {
      Random rnd = new Random();
      BigInteger e32Bit;

      // Until e <= 1 or e > phiN or gcd(e, phiN) != 1
      do {
```

```
      e32Bit = new BigInteger(32, rnd);
   } while((e32Bit.compareTo(BigInteger.ONE) <= 0 ||
        e32Bit.compareTo(phiN64Bit) >= 0) ||
         !e32Bit.gcd(phiN64Bit).equals(BigInteger.ONE) ||
         e32Bit.equals(phiN64Bit.add(BigInteger.TWO)
         .divide(BigInteger.TWO)));

   return e32Bit;
}

/**
 * Initialize an RSA Object that can be used to Encrypt and Decrypt
     messages
 * with random (and odds) p and q.
 */
public RSA() {
   Random rnd = new Random();

   // Generate random p and q
   do {
      p32Bit = BigInteger.probablePrime(32, rnd);
      q32Bit = BigInteger.probablePrime(32, rnd);
   } while(p32Bit.equals(q32Bit));

   initializer(p32Bit, q32Bit);
}

/**
 * Initialize an RSA Object, with the specified exponent e, that can
     be used to
 * Encrypt and Decrypt messages with random (and odds) p and q.
 * @param e32Bit
 */
public RSA(BigInteger e32Bit) {
   if(e32Bit == null)
      throw new NullPointerException();
   if(e32Bit.compareTo(BigInteger.ONE) <= 0)
      throw new IllegalArgumentException();

   Random rnd = new Random();

   // Generate random p and q
   do {
      p32Bit = BigInteger.probablePrime(32, rnd);
      q32Bit = BigInteger.probablePrime(32, rnd);

      // Compute n = p * q
      n64Bit = p32Bit.multiply(q32Bit);

      // Compute phiN = (p - 1) * (q - 1)
```

```
      phiN64Bit = p32Bit.subtract(BigInteger.ONE)
      .multiply(q32Bit.subtract(BigInteger.ONE));
   } while(p32Bit.equals(q32Bit) ||
       !e32Bit.gcd(phiN64Bit).equals(BigInteger.ONE) ||
        e32Bit.compareTo(phiN64Bit) >= 0 ||
        e32Bit.equals(phiN64Bit.add(BigInteger.TWO)
        .divide(BigInteger.TWO)));

   this.e32Bit = e32Bit;

   // Compute d = e ^ (-1) mod phiN
   privateKey = this.e32Bit.modInverse(phiN64Bit);
}

/**
 * Initialize an RSA Object, with the specified values of p and q,
     that can be used to
 * Encrypt and Decrypt messages with random (and odds) p and q.
 * @param p32Bit
 * @param q32Bit
 */
public RSA(BigInteger p32Bit, BigInteger q32Bit) {
   if(p32Bit == null || q32Bit == null)
      throw new NullPointerException();
   if(!p32Bit.isProbablePrime(100) || !q32Bit.isProbablePrime(100) ||
       p32Bit.equals(q32Bit))
      throw new IllegalArgumentException();

   this.p32Bit = p32Bit;
   this.q32Bit = q32Bit;

   initializer(this.p32Bit, this.q32Bit);
}

/**
 * Given a message, it encrypts it
 * @param msg
 * @return Encrypted message
 */
public BigInteger encrypt(BigInteger msg) {
   if(msg == null)
      throw new NullPointerException();
   if(msg.compareTo(n64Bit) > 0)
      throw new IllegalArgumentException("Msg too long");

   return msg.modPow(e32Bit, n64Bit);    // (msg ^ e) mod n
}

/**
 * Given a cryptogram, it decipher it
```

```
     * @param c : cryptogram
     * @return Deciphered message
     */
    public BigInteger decrypt(BigInteger c) {
        if(c == null)
            throw new NullPointerException();

        return c.modPow(privateKey, n64Bit);  // (c ^ privateKey) mod n
    }

    /**
     * Print p, q, n, phiN, public key and private key
     */
    public void printInfo() {
        System.out.println("p : " + p32Bit);
        System.out.println("q : " + q32Bit);
        System.out.println("phiN : " + phiN64Bit + System.lineSeparator());

        System.out.println("public key <" + e32Bit + ", " + n64Bit + ">");
        System.out.println("private key <" + privateKey + ">" +
            System.lineSeparator());
    }

    /**
     * Return the first member of the public key
     * @return e32Bit
     */
    public BigInteger getExponent() {
        return e32Bit;
    }

    /**
     * Return the second member of the public key
     * @return n64Bit
     */
    public BigInteger getN() {
        return n64Bit;
    }
}
```

B Codice - RsaAttacker

```
/**
 * ########################################################
 *
 * @author: Michael De Angelis
 * @mat: 560049
 * @project: Esperienze di Programmazione [ESP]
 * @AA: 2019 / 2020
 *
 * ########################################################
 */

package rsa_attacks;

import java.math.BigInteger;
import java.util.ArrayList;

public final class RsaAttacker {
   private RsaAttacker() {};

   private static BigInteger MAX = new BigInteger("4294967296");

   /**
    * Given the public key, it performs a brute force
    * attack in search of the private key factoring n.
    * @param e32Bit
    * @param n64Bit
    * @return Time spent
    */
   public static long bruteForceAttack(BigInteger e32Bit, BigInteger
       n64Bit) {
      if(e32Bit == null || n64Bit == null)
         throw new NullPointerException();

      System.out.println(System.lineSeparator() + "Brute Force : " +
          System.lineSeparator());

      long start = System.currentTimeMillis();

      // Looking for p or q
      BigInteger factor = new BigInteger("1");
      for(BigInteger i = new BigInteger("2"); i.compareTo(MAX) < 0; i =
          i.add(BigInteger.ONE)) {
         if(n64Bit.mod(i).compareTo(BigInteger.ZERO) == 0) {
            factor = i;
            break;
         }
      }
```

```
    // Gets the other factor
    BigInteger factor2 = n64Bit.divide(factor);

    // Compute phiN
    BigInteger phiN =
        factor.subtract(BigInteger.ONE).multiply(factor2.subtract(BigInteger.ONE));

    // Compute the private key
    BigInteger privateKey = e32Bit.modInverse(phiN);

    long timePassed = System.currentTimeMillis() - start;

    System.out.println("Completed in " + timePassed + " ms");
    System.out.println("Private Key : <" + privateKey + ">" +
        System.lineSeparator());

    return timePassed;
}

/**
 * Given n, check if is a perfect square
 * @param n
 * @return true if n is a perfect square, false otherwise
 */
private static boolean isPerfectSquare(BigInteger n) {
    BigInteger sqrt = n.sqrt();

    if(sqrt.multiply(sqrt).equals(n) ||
        sqrt.add(BigInteger.ONE).multiply(sqrt.add(BigInteger.ONE)).equals(n))
       return true;

    return false;
}

/**
 * Given the public key, product of two close values, p and q,
 * apply the Fermat factoring algorithm to find the private key.
 * @param e32Bit
 * @param n64Bit
 * @return Time spent
 */
public static long fermatFactoringAttack(BigInteger e32Bit,
     BigInteger n64Bit) {
    if(e32Bit == null || n64Bit == null)
       throw new NullPointerException();

    System.out.println(System.lineSeparator() + "Fermat Factoring :" +
        System.lineSeparator());
```

```
   long start = System.currentTimeMillis();

   // Let k be the smallest positive integer so that k^2 > n
   BigInteger z = n64Bit.sqrt().add(BigInteger.ONE);

   // Find the w such that z ^ 2 - n = w ^ 2
   BigInteger w = null;
   while(true) {
      w = z.multiply(z).subtract(n64Bit);
      if(isPerfectSquare(w))
         break;

      z = z.add(BigInteger.ONE);
   }
   w = w.sqrt();

   // Gets the factors p and q
   BigInteger p = z.add(w);
   BigInteger q = z.subtract(w);

   // Compute phiN
   BigInteger phiN =
       p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));

   // Compute the private key
   BigInteger privateKey = e32Bit.modInverse(phiN);

   long timePassed = System.currentTimeMillis() - start;

   System.out.println("Completed in " + timePassed + " ms");
   System.out.println("Private Key : <" + privateKey + ">" +
       System.lineSeparator());

   return timePassed;
}

/**
 * Calculate the nthroot of the given BigInteger
 * @param n @param x
 * @return nthroot of x
 */
private static BigInteger nthRoot(int n, BigInteger x) {
    BigInteger y = BigInteger.ZERO;
    for(int m = (x.bitLength() - 1) / n; m >= 0; --m) {
        BigInteger z = y.setBit(m);
        int cmp = z.pow(n).compareTo(x);
        if(cmp == 0)
         return z; // found exact root
        if(cmp < 0)
         y = z;    // keep bit set
```

```
    }

    return y; // return floor of exact root
}

/**
 * Given a (little) value e and e user that received the same
     message; use the Chinese rest
 * theorem to find the only m' < n such that m' congruous m ^ e mod n.
 * @param e32Bit
 * @param n64Bits
 * @param cmsgs
 * @return Time spent
 */
public static long eSameValueAttack(int e32Bit, ArrayList<BigInteger>
    n64Bits, ArrayList<BigInteger> cmsgs) {
   if(n64Bits == null || cmsgs == null)
      throw new NullPointerException();
   if(e32Bit <= 1 || n64Bits.size() != e32Bit || cmsgs.size() !=
       e32Bit)
      throw new IllegalArgumentException();

   System.out.println(System.lineSeparator() + "Same Exponent Attack
       :" + System.lineSeparator());

   long start = System.nanoTime();

   // Compute n = n1 * n2 * ... * n_e
   BigInteger nChineseTh = BigInteger.ONE;
   for(BigInteger x : n64Bits)
      nChineseTh = nChineseTh.multiply(x);

   // Find m' congrous m ^ e mod n with the Chinese rest theorem
   BigInteger sum = BigInteger.ZERO;
   for(int i = 0; i < n64Bits.size(); i++) {
      BigInteger p = nChineseTh.divide(n64Bits.get(i));
      BigInteger tmp = p.modInverse(n64Bits.get(i));
      sum = sum.add(cmsgs.get(i).multiply(tmp).multiply(p));
   }

   BigInteger msg = nthRoot(e32Bit, sum.mod(nChineseTh));

   long timePassed = System.nanoTime() - start;

   System.out.println("Completed in " + timePassed + " ns");
   System.out.println("Message Discovered : <" + msg + ">" +
       System.lineSeparator());

   return timePassed;
}
```

```
/**
 * Private class that represent a simple triple
 */
private static class Triple {
   private final BigInteger d;
   private final BigInteger s;
   private final BigInteger t;

   // Constructor
   private Triple(final BigInteger d, final BigInteger s, final
       BigInteger t) {
      if(d == null || s == null || t == null)
        throw new NullPointerException();

      this.d = d;
      this.s = s;
      this.t = t;
   }

   private final BigInteger getD() {
      return d;
   }

   private final BigInteger getS() {
      return s;
   }

   private final BigInteger getT() {
      return t;
   }
}

/**
 * Compute the Extended Euclidean Algorithm with the given value a
     and b
 * @param a @param b
 * @return Triple that represent the value of the Extended Euclidean
     Algorithm
 */
private static Triple apply(final BigInteger a, final BigInteger b) {
   if(b.equals(BigInteger.ZERO))
      return new Triple(a, BigInteger.ONE, BigInteger.ZERO);
   else {
      final Triple extension = apply(b, a.mod(b));
      return new Triple(extension.getD(), extension.getT(),
      extension.getS().subtract(a.divide(b)
      .multiply(extension.getT())));
   }
}
```

```
  /**
   * Given 2 users that received the same message with the same value
       of n; use c1 ^ s * c2 ^ t mod n
   * to find the the message m.
   * @param n64Bit
   * @param e32BitU1 @param e32BitU2
   * @param c1 @param c2
   * @return Time spent
   */
  public static long nSameValueAttack(BigInteger n64Bit, BigInteger
      e32BitU1, BigInteger e32BitU2, BigInteger c1, BigInteger c2) {
    if(n64Bit == null || e32BitU1 == null || e32BitU2 == null || c1 ==
        null || c2 == null)
      throw new NullPointerException();
    if(!e32BitU1.gcd(e32BitU2).equals(BigInteger.ONE))
      throw new IllegalArgumentException();

    System.out.println(System.lineSeparator() + "Same n Attack :" +
        System.lineSeparator());

    long start = System.nanoTime();

    // Gets the s and t such that s * e32BitU1 + t * e32BitU2 = 1
    Triple st = apply(e32BitU1, e32BitU2);

    // Compute c1 ^ s * c2 ^ t mod n
    BigInteger msg = null;
    if(st.getS().compareTo(BigInteger.ZERO) < 0) {
      BigInteger tmp = c1.modInverse(n64Bit);
      msg = tmp.modPow(st.getS().negate(),
          n64Bit).multiply(c2.modPow(st.getT(), n64Bit)).mod(n64Bit);
    }
    else {
      BigInteger tmp = c2.modInverse(n64Bit);
      msg = c1.modPow(st.getS(),
          n64Bit).multiply(tmp.modPow(st.getT().negate(),
          n64Bit)).mod(n64Bit);
    }

    long timePassed = System.nanoTime() - start;

    System.out.println("Completed in " + timePassed + " ns");
    System.out.println("Message Discovered : <" + msg + ">" +
        System.lineSeparator());

    return timePassed;
  }
}
```

C Codice - bruteForceTesting

```
/**
 * ########################################################
 *
 * @author: Michael De Angelis
 * @mat: 560049
 * @project: Esperienze di Programmazione [ESP]
 * @AA: 2019 / 2020
 *
 * ########################################################
 */

package rsa_attacks;

public class bruteForceTesting {

   public static void main(String[] args) {
      // Perform a set of 10 brute force attacks over 10 differet RSA
          Objects.
      for(int i = 0; i < 10; i++) {
         // Rsa object used to perform the brute force attack
         RSA rsaObj = new RSA();
         rsaObj.printInfo();

         RsaAttacker.bruteForceAttack(rsaObj.getExponent(),
             rsaObj.getN());
      }
   }
}
```

D Codice - fermatFactoringTesting

```java
/**
 * ########################################################
 *
 * @author: Michael De Angelis
 * @mat: 560049
 * @project: Esperienze di Programmazione [ESP]
 * @AA: 2019 / 2020
 *
 * ########################################################
 */

package rsa_attacks;

public class fermatFactorizationTesting {
   public static void main(String[] args) {
      if(args.length != 1) {
         System.out.println("Usage fermatFactorizationTesting <nTry>");
         System.exit(-1);
      }

      int nTry = Integer.parseInt(args[0]);

      // Simulate a Fermat Factoring attack with nTry random RSA object
      long fermatFactoringTimeRqst = 0;
      for(int i = 0; i < nTry; i++) {
         RSA rsaObjFF = new RSA();
         rsaObjFF.printInfo();
         fermatFactoringTimeRqst +=
             RsaAttacker.fermatFactoringAttack(rsaObjFF.getExponent(),
             rsaObjFF.getN());
      }

      System.out.println("Fermat Factoring Attack completed in " +
          fermatFactoringTimeRqst + " ms");
   }
}
```

E Codice - eSameValueTesting

```java
/**
 * ########################################################
 *
 * @author: Michael De Angelis
 * @mat: 560049
 * @project: Esperienze di Programmazione [ESP]
 * @AA: 2019 / 2020
 *
 * ########################################################
 */

package rsa_attacks;

import java.math.BigInteger;
import java.util.ArrayList;
import java.util.Random;

public class eSameValueTesting {

   public static void main(String[] args) {
      if(args.length != 1) {
         System.out.println("Usage eSameValueTesting <nTry>");
         System.exit(-1);
      }

      long start = System.currentTimeMillis();
      int nTry = Integer.parseInt(args[0]);

      // Simulate an exponent based attack with e = 3;
      for(int i = 0; i < nTry; i++) {
         int e = 3;
         ArrayList<RSA> rsaObjs = new ArrayList<RSA>();
         BigInteger ee = new BigInteger("3");
         // Create e user with the same exponent and different value of
             n = p * q
         for(int j = 0; j < e; j++) {
            rsaObjs.add(new RSA(ee));
            rsaObjs.get(j).printInfo();
         }

         // Create the list of the components n of the public key of
             each users
         ArrayList<BigInteger> n = new ArrayList<BigInteger>();
         for(RSA rsa : rsaObjs)
            n.add(rsa.getN());
```

```
        // Get the minimum n in order to create a message < n (for the
            modulo reduction)
        BigInteger min = n.get(0);
        for(int j = 1; j < n.size(); j++)
           if(n.get(j).compareTo(min) < 0)
              min = n.get(j);

        // Generate a random message less that min
        Random rnd = new Random();
        BigInteger msg;
        do {
           msg = new BigInteger(32, rnd);
        } while(msg.compareTo(min) >= 0);

        // Create the list of the cryptograms generated by each user
            for the same message msg
        ArrayList<BigInteger> c = new ArrayList<BigInteger>();
        for(RSA rsa : rsaObjs)
           c.add(rsa.encrypt(msg));

        System.out.println("Original Random Message: " + msg +
            System.lineSeparator());

        RsaAttacker.eSameValueAttack(e, n, c);
     }

     System.out.println("Same Exponent Attack completed in " +
         (System.currentTimeMillis() - start) + " ms");
  }
}
```

F Codice - nSameValueTesting

```java
/**
 * ########################################################
 *
 * @author: Michael De Angelis
 * @mat: 560049
 * @project: Esperienze di Programmazione [ESP]
 * @AA: 2019 / 2020
 *
 * ########################################################
 */

package rsa_attacks;

import java.math.BigInteger;
import java.util.Random;

public class nSameValueTesting {

   public static void main(String[] args) {
      if(args.length != 1) {
         System.out.println("Usage nSameValueTesting <nTry>");
         System.exit(-1);
      }

      long start = System.currentTimeMillis();
      int nTry = Integer.parseInt(args[0]);

      // Simulate an n based attack
      for(int i = 0; i < nTry; i++) {
         // Generate p and q such that p != q
         Random rnd = new Random();
         BigInteger p32Bit = null;
         BigInteger q32Bit = null;
         do {
            p32Bit = BigInteger.probablePrime(32, rnd);
            q32Bit = BigInteger.probablePrime(32, rnd);
         } while(p32Bit.equals(q32Bit));

         // Generate two RSA object with the same value of n and
             different exponents; gcd(e1, e2) must be 1
         RSA rsa1 = null;
         RSA rsa2 = null;
         do {
            rsa1 = new RSA(p32Bit, q32Bit);
            rsa2 = new RSA(p32Bit, q32Bit);
         } while(rsa1.getExponent().equals(rsa2.getExponent()) ||
               !rsa1.getExponent().gcd(rsa2.getExponent())
```

```
                .equals(BigInteger.ONE));

        rsa1.printInfo();
        rsa2.printInfo();

        // Generate a random message less that rsa1.N and rsa2.N
        BigInteger msg;
        do {
           msg = new BigInteger(32, rnd);
        } while(msg.compareTo(rsa1.getN()) >= 0);

        System.out.println("Original Random Message: " + msg +
            System.lineSeparator());

        RsaAttacker.nSameValueAttack(rsa1.getN(), rsa1.getExponent(),
            rsa2.getExponent(), rsa1.encrypt(msg), rsa2.encrypt(msg));
     }

     System.out.println("Same n Attack completed in " +
         (System.currentTimeMillis() - start) + " ms");
  }
}
```

References

[1] F. Luccio A. Bernasconi P. Ferragina. "Elementi di Crittografia". In: Manuali. Pisa University Press, 2015. Chap. 8.4. ISBN: 8867414607.

[2] Wikipedia. *Algoritmo esteso di Euclide*. URL: https://it.wikipedia.org/wiki/Algoritmo_esteso_di_Euclide.

[3] Wikipedia. *Identità di Bézout*. URL: https://it.wikipedia.org/wiki/Identit%C3%A0_di_B%C3%A9zout.

[4] Wikipedia. *Metodo forza bruta*. URL: https://it.wikipedia.org/wiki/Metodo_forza_bruta.

[5] Wikipedia. *Numero primo di Mersenne*. URL: https://it.wikipedia.org/wiki/Numero_primo_di_Mersenne.

[6] Wikipedia. *Pseudoprimo di Eulero*. URL: https://it.wikipedia.org/wiki/Pseudoprimo_di_Eulero.

[7] Wikipedia. *Teorema cinese del resto*. URL: https://it.wikipedia.org/wiki/Teorema_cinese_del_resto.

[8] Wikipedia. *Test di Lucas-Lehmer*. URL: https://it.wikipedia.org/wiki/Test_di_Lucas-Lehmer.

[9] Wikipedia. *Test di Miller-Rabin*. URL: https://it.wikipedia.org/wiki/Test_di_Miller-Rabin.