Politecnico di Torino

Corso di Laurea
in Matematica per l'Ingegneria

Tesi di Laurea

ANALISI DELLA VULNERABILITÀ ROCA SU RSA

Relatore:
Prof. Bazzanella Danilo

Correlatore:
Dott. Rossi Matteo

Candidato:
Torasso Pablo

Anno Accademico 2020/2021

La matematica è l'alfabeto
nel quale Dio
ha scritto l'universo

Galileo Galilei

Abstract

La sicurezza informatica riveste un ruolo fondamentale nella società moderna, in quanto protegge i dati sensibili scambiati tra i vari dispositivi elettronici. L'algebra modulare assicura la possibilità di trasmettere informazioni in modo sicuro attraverso la crittografia RSA, che attualmente è lo standard utilizzato dalla maggior parte delle aziende per la protezione dati. La presenza di falle in questo sistema di trasmissione può condurre all'acquisizione di informazioni sensibili e al loro sfruttamento a fini illegali.

Il focus principale di questa trattazione è analizzare da un punto di vista matematico la crittografia RSA e in particolare la libreria RSALib, sviluppata dall'azienda Infineon Technologies. Ci si prefigge inoltre di analizzare i punti di forza e i punti di debolezza della libreria, dimostrando infine l'impraticabilità della stessa nella sicurezza informatica.

Indice

Capitolo 1

Introduzione

RSA è un algoritmo largamente diffuso per la crittografia asimmetrica usata per le firme digitali e la cifratura dei messaggi. La sicurezza di RSA è basata sulla complessità computazionale della fattorizzazione in numeri primi.

La parte privata di una chiave è un oggetto sensibile. Un utente potrebbe usare hardware come smartcard crittografiche per memorizzare e usare il valore della chiave privata. Attacchi portati a buon fine contro RSA danno la possibilità di impersonificare il proprietario della chiave e decriptare il messaggio privato. Le chiavi usate da hardware e software sicuri sono di speciale interesse poichè proteggono informazioni di grande valore, per esempio dati di transazioni.

L'utilizzo della crittografia RSA richiede due grandi numeri primi, p e q. I grandi numeri primi, p e q, vengono generati seguendo algoritmi ben precisi, conferendo alla chiave privata proprietà diverse in base all'algoritmo usato; per esempio, una smartcards con poca potenza di calcolo utilizzerà algoritmi leggeri per velocizzare l'esecuzione. La trattazione dimostrerà come tali chiavi possono risultare insicure.

La vulnerabilità ROCA (acronimo di "Return of Coppersmith's attack"), nota con il codice identificativo CVE-2017-15361, è una debolezza crittografica che permette di recuperare facilmente la chiave privata dalla chiave pubblica generata da devices che sfruttano la libreria RSALib per la generazione della chiave. Il team di ricercatori che ha scoperto la vulnerabilità stima che milioni di smart cards siano affette dalla vulnerabilità ROCA[1].

Il matematico statutinitense Coppersmith ha dimostrato che la conoscenza o il recupero di tutti i bits di una chiave privata non è necessario per un attacco di successo. Se almeno metà dei bits di uno dei due primi è conosciuto, i rimanenti bits possono essere computazionalmente recuperati.

In questa tesi verrà analizzato il metodo di generazione delle chiavi della libreria RSALib e le sue vulnerabilità.

Capitolo 2

RSA: Crittografia a chiave pubblica

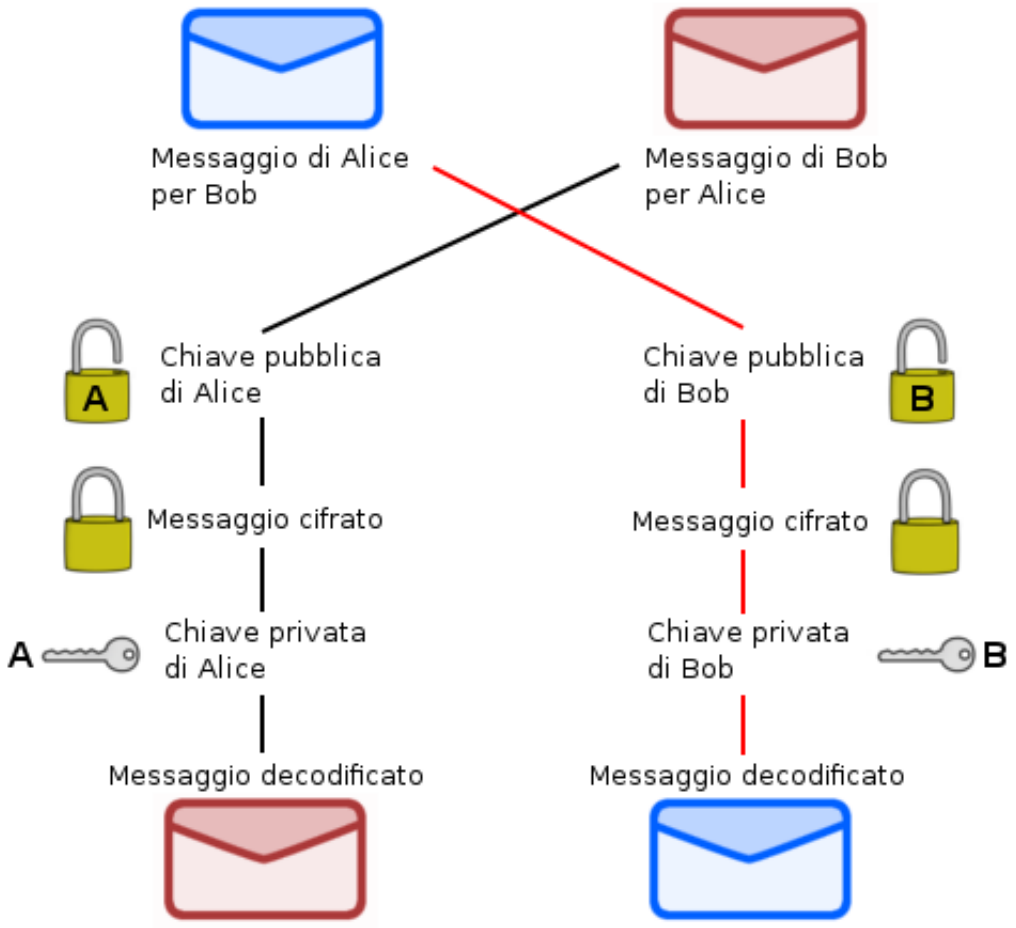
In crittografia la sigla RSA indica un algoritmo di crittografia asimmetrica (detta anche a chiave pubblica) utilizzabile per cifrare o firmare informazioni, inventato nel 1977 da Ronald Rivest, Adi Shamir e Leonard Adleman. In questo capitolo si cerca di presentare le generalità dell'argomento e in seguito, più in dettaglio, i fondamenti matematici su cui si basa la crittografia RSA.

2.1 Crittografia a chiave pubblica

La crittografia a chiave pubblica è un tipo di crittografia dove ad ogni attore coinvolto nella comunicazione è associata una coppia di chiavi:
La chiave pubblica, che deve essere distribuita;
La chiave privata, personale e segreta.

Tale struttura permette di evitare qualunque problema connesso alla necessità di uno scambio in modo sicuro dell'unica chiave utile alla cifratura/decifratura presente invece nella crittografia simmetrica. Il concetto principale di questo tipo di sistema è il seguente: con la chiave pubblica si cifra un messaggio e solamente con la chiave privata è possibile decifrare il messaggio. Ci sono due funzioni che possono essere realizzate: cifrare messaggi con la chiave pubblica per garantire che solo il titolare della chiave privata possa decifrarlo oppure usare la chiave pubblica per autenticare un messaggio inviato dal titolare con la chiave privata abbinata.
Per fare ciò, deve essere computazionalmente facile per un utente generare una coppia di chiavi, pubblica e privata, da utilizzare per cifrare e decifrare. La forza di un sistema di crittografia a chiave pubblica si basa sulla difficoltà di determinare la chiave privata a partire dalla chiave pubblica.

Figura 2.1: Esempio di utilizzo della crittografia RSA



[2]

La sicurezza dipende quindi solo dal mantenere la chiave privata segreta, mentre la chiave pubblica può essere pubblicata senza compromettere la sicurezza.
I sistemi di crittografia a chiave pubblica sono costruiti su algoritmi basati su problemi matematici che attualmente non ammettono alcuna soluzione efficiente; per esempio quelli che riguardano la fattorizzazione di un numero intero, il logaritmo discreto e certe operazioni sulle curve ellittiche.
A causa del peso computazionale della crittografia asimmetrica, essa di solito è usata solo per piccoli blocchi di dati, in genere il trasferimento di una chiave di cifratura simmetrica, per esempio una chiave di sessione. La chiave simmetrica è utilizzata per cifrare messaggi lunghi. La cifratura/decifratura simmetrica è basata su algoritmi semplici ed è molto più veloce.

Gli algoritmi a chiave pubblica sono ingredienti fondamentali della sicurezza dei crittosistemi, applicazioni e protocolli. Essi sono alla base dei diversi standard Internet, come ad esempio Transport Layer Security (TLS), S/MIME, PGP e GPG. Alcuni algoritmi a chiave pubblica forniscono una distribuzione di chiave e segretezza, ad esempio lo scambio di chiavi Diffie-Hellman, alcuni forniscono firme digitali , altri forniscono entrambe, come l'algoritmo RSA.

2.2 Teoria matematica alla base di RSA

Scelta una determinata lunghezza in bits della chiave, il processo per costruire una chiave pubblica e una chiave privata è il seguente:

1. Vengono generati due numeri primi p e q.
2. Viene calcolato $N = p \cdot q$, chiamato modulo, e il prodotto $\phi(N) = (p-1) \cdot (q-1)$, funzione di Eulero.
3. Viene scelto poi un numero e, chiamato esponente pubblico, coprimo con $\phi(N)$ e più piccolo di $\phi(N)$.
4. Viene calcolato il numero d, chiamato esponente privato, tale che il suo prodotto con e sia congruo a 1 modulo $\phi(N)$, ovvero che $ed \equiv 1 \bmod \phi(N)$.

La chiave pubblica è (N, e) , mentre la chiave privata è (p, q, d).

La sicurezza dell'algoritmo risiede nel impossibilità di calcolare l'esponente privato dalla conoscenza dell'esponente pubblico, infatti non basta la conoscenza di N ma serve ϕ(N) = (p − 1) · (q − 1) e il suo calcolo richiede tempi molto elevati.

Un messaggio m viene cifrato attraverso l'operazione $m^e \bmod N$ trasformandolo nel messaggio cifrato c. Una volta trasmesso, c viene decifrato con $c^d \bmod N = m$. Il procedimento funziona solo se $m < N$ e la chiave e utilizzata per cifrare e la chiave d utilizzata per decifrare sono legate tra loro dalla relazione $ed \equiv 1 \bmod \varphi(N)$; dal momento che un messaggio viene cifrato utilizzando la chiave pubblica, esso puo essere decifrato soltanto usando la chiave privata associata alla chiave pubblica.

La firma digitale è l'utilizzo inverso delle chiavi: il messaggio viene crittografato con la chiave privata, in modo che chiunque possa, utilizzando la chiave pubblica, decifrarlo e, oltre a poter leggere in chiaro, essere certo che il messaggio possa essere stato mandato solamente dal possessore della chiave privata. Per motivi di efficienza e comodità, di norma viene inviato il messaggio in chiaro con allegata la firma digitale di un hash del messaggio stesso; in questo modo il ricevente può direttamente leggere il messaggio, che è in chiaro, utilizzare la chiave pubblica per estrarre l'hash dalla firma e verificare che questo sia uguale a quello calcolato localmente sul messaggio ricevuto. Se l'hash utilizzato è crittograficamente sicuro, la corrispondenza dei due valori conferma che il messaggio ricevuto è identico a quello originalmente firmato e trasmesso.

La decifratura del messaggio è assicurata grazie ad un teorema; infatti sia m il messaggio e $c = m^e$ il messaggio cifrato, per il teorema di Fermat-Eulero:

$$c^d \bmod N \equiv m^{ed} \bmod N \equiv m^{\phi(N)} \equiv m \bmod N$$

Capitolo 3

La libreria *RSALib*

In questo capitolo ci soffermeremo ad analizzare la struttura matematica delle chiavi generate dalla libreria RSALib, analizzando a fondo le proprietà che ne conseguono.

3.1 Generazione dei primi

Per la generazione del modulo N abbiamo bisogno di due primi p e q:

$$N = p \cdot q$$

Nel caso della libreria *RSALib*, i primi hanno una forma matematica precisa; ogni numero primo p, q ha la seguente forma:

$$p = k \cdot M + (65537^a \bmod M) \tag{3.1}$$

i parametri $k, a \in \mathbb{N}$ sono sconosciuti.
$M \in \mathbb{N}$ è un primoriale (prodotto dei primi n numeri primi successivi):

$$M = \prod_{i=1}^{n} P_i = 2 \cdot 3 \cdot 5 \cdot 7 ... \cdot P_n$$

Il valore di M dipende dalla grandezza della chiave da generare. Chiavi più grandi saranno generate da primoriali più grandi. Per esempio, chiavi di lunghezza compresa tra 512 e 960 bits avranno $n = 39 \rightarrow M = \prod_{i=1}^{39} P_i$
La seguente tabella definisce i valori dei primoriali per varie lunghezze di bits della chiave:

Lunghezza chiave	n	M
512 - 960	39	P_{39}
992 - 1952	71	P_{71}
1984 - 3936	126	P_{126}
3968 - 4096	225	P_{225}

3.2 Caratteristiche dell'algoritmo

Tale generazione delle chiavi comporta determinate caratteristiche matematiche; alcune di queste sono la causa per cui la tipologia di chiavi generata da *RSALib* soffre di criticità importanti e può essere attaccata facilmente.

3.2.1 L'esponente pubblico 65537

L'esponente pubblico utilizzato dalla libreria RSALib e=65537 è uno standart all'interno della crittografia RSA.
Il numero primo 65537 appartiene ad una categoria particolare di numeri primi, numeri di Fermat, insieme ai primi 3, 5, 17 e 257; Infatti questi cinque numeri primi hanno la caratteristica unica di avere una scrittura binaria formata da pochi 1 e tanti 0, cioè sono della forma $2^n + 1$ per certi $n \in N$.

Decimale	Binario
3	11
5	101
17	10001
257	100000001
65537	10000000000000001

L'operazione $65537^a \bmod M$, esponenziale modulo M, è compiuta attraverso il metodo di calcolo binario *exponential by squaring* che compie operazioni di calcolo solo in corrispondenza degli 1.
La scelta di un esponente pubblico con basso impatto di calcolo garantisce che questo algoritmo possa essere usato anche su dispositivi con potenza di calcolo minima.
Il numero primo 65537, quindi, è il giusto compromesso tra un numero primo abbastanza grande da garantire sicurezza ed un numero in grado di velocizzare la generazione delle chiavi.

3.2.2 Entropia dei numeri primi

Il problema principale della libreria RSALib nella generazione dei primi è che M è grande rispetto a p e q.
Per una grandezza della chiave di 512 bits, p, q sono primi di 256 bits e M è di grandezza 219 bits; questo vuol dire che k ha un'entropia di $256 - 219 = 37$ bits.
Per calcolare il limite superiore di a, dobbiamo calcolare il numero di elementi k presenti nel gruppo moltiplicativo creato dal generatore g modulo M.

$$\{1, g, g^2, g^3, ..., g^{k-1}\} \bmod M$$

In altre parole, dobbiamo calcolare l'ordine moltiplicativo di g modulo M, cioè il più piccolo intero k tale che $g^k = 1 \bmod M$.
Nel nostro caso $g = 65537$.

Per chiavi RSA di 512 bit, a ha un entropia di 62 bits; quindi, il risultante numero primo p non ha un entropia di 256 bits bensì di $37 + 62 = 99$ bits.
Questo significa che al posto di esistere 2^{256} possibilità per la chiave p, ne esistono solo 2^{99}

3.2.3 Riconoscimento della chiave

Un'altro grosso problema della libreria RSALib è l'esistenza di un metodo rapido e semplice per riconoscere se la chiave sia stata generata dalla libreria.

Analiziamo la forma di N:

$$N = p \cdot q = (k \cdot M + 65537^a \bmod M) \cdot (l \cdot M + 65537^b \bmod M) \tag{3.2}$$

con $a, b, k, l \in \mathbb{N}$.
L'identità precedente implica

$$N \equiv 65537^{a+b} \equiv 65537^c \bmod M$$

con $c \in \mathbb{N}$.
L'esistenza del logaritmo discreto $c = log_{65537} N \bmod M$ è usato come fingerprint del modulo pubblico N generato dalla libreria RSALib.

Nonostante il problema del logaritmo discreto sia un problema generalmente difficile da risolvere, nel nostro caso può essere facilmente calcolato usando l'algoritmo di Pohlig-Hellman.
L'algoritmo viene usato in modo efficente per calcolare logaritmi discreti di un gruppo G, la cui grandezza $|G|$ è uno *smooth number* (composto solo da fattori piccoli).
Questo è esattamente il caso del gruppo $G = [65537]$, sottogruppo di $\mathbb{Z}_M^*$ generato da 65537.
La grandezza di G è uno *smooth number*, per esempio $G = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 23 \cdot 29 \cdot 37 \cdot 41 \cdot 53 \cdot 83$ per chiavi di 512 bits.

3.2.4 Falsi positivi

L'esistenza del logaritmo discreto viene usato come forte metodo di fingerprinting del modulo pubblico N.
La ragione di ciò è che mentre i primi p, q e i moduli N generati casualmente appartengono all'intero gruppo $\mathbb{Z}_M^*$, i numeri primi e i moduli generati dalla libreria *RSALib* appartengono al gruppo G, un piccolo sottogruppo di $\mathbb{Z}_M^*$.
La probabilità che un modulo casuale N di 512 bit appartenga al sottogruppo G è $2^{62-216} = 2^{-154}$. Questa probabilità è ancora più piccola per chiavi più grandi.
Possiamo affermare con sicurezza la seguente conclusione:
una chiave RSA è stata generata dalla libreria RSALib se e solo se l'algoritmo di Pohlig-Hellman risolve il logaritmo discreto $log_{65537} N \bmod M$.
Questa aspettativa teorica è stata confermata in modo sperimentale[1].

Capitolo 4

Principi dell'attacco

L'attacco è basato sull'algoritmo di Coppersmith rivisitato da Howgrave-Graham. Questo algoritmo è spesso usato per condurre attacchi su modelli rilassati RSA, cioè quando si è già in possesso di informazione parziale.

4.1 Procedimento generale

L'algoritmo di Coppersmith è un algoritmo in grado di ricavare p e q nel caso in cui parte dei bit siano già conosciuti. Analizzando la struttura di p e q:

$$p = k \cdot M + (65537^a \bmod M)$$

i valori $a, k \in \mathbb{N}$ sono sconosciuti.
$M \in \mathbb{N}$ è ricavabile dalla lunghezza del modulo pubblico; calcolando la lunghezza di N, conosciamo la lunghezza dei due numeri primi: $|p|, |q| = \frac{|N|}{2}$ con |p|, |q| e |N| lunghezza in bit di p, q e N ; di conseguenza è noto il valore n con cui si ricava il primoriale M.

Ipotizzando di conoscere a, iterando sui possibili valori di esso, $(65537^a \bmod M)$ è la parte nota, e possiamo utilizzare il metodo di Coppersmith su

$$p = k \cdot M + (65537^a \bmod M)$$

con $(65537^a \bmod M)$ bits noti.

4.2 Problema equivalente

Prima di poter procedere con la risoluzione, è necessario riscrivere il problema.
Per poter utilizzare l'algoritmo di Coppersmith-Howgrave-Graham, dobbiamo trasformare il problema in una equazione polinomiale modulare $f(x) \equiv 0 \bmod p$ con p divisore ignoto di N e la radice $x_0 \rightarrow f(x_0) \equiv 0 \bmod p$ incognita dell'algoritmo.

Partendo dalla forma iniziale di p (3.1), possiamo riscrivere:

$$f(x) = x \cdot M + (65537^a \bmod M) \mod N$$

L'algoritmo cercherà la radice piccola $x_0 = k$ tale che $f(x_0) \equiv 0 \bmod p$.

L'ultimo passo è rendere il polinomio monico, cioè rendere il coefficente di grado massimo 1; possiamo ottenere il polinomio monico, moltiplicando per l'inverso di M modulo N:

$$\begin{aligned} f(x) &= x \cdot M \cdot (M^{-1} \bmod N) + (65537^a \bmod M) \cdot (M^{-1} \bmod N) \\ &= x + (65537^a \bmod M) \cdot (M^{-1} \bmod N) \mod N \end{aligned}$$

4.3 Algoritmo di Coppersmith

L'algoritmo di Coppersmith è più lento quando si usano i bits minimi conosciuti, cioè la metà dei bits del modulo pubblico; con più bits conosciuti, il tempo di esecuzione diminuisce.

4.3.1 Algoritmo Naive

Con il termine naive si tende ad indicare un algoritmo, efficace ma non ottimizzato, nel risolvere un dato problema. Nel caso del nostro problema affronteremo più avanti alcuni tipi di ottimizzazioni possibili.

Sia N della forma (3.2), cerchiamo i fattori p e q. Per trovare p e q bisogna prima trovare i due parametri k, a corretti. L'algoritmo naive itera sui possibili valori di $65537^a \bmod M$, che dipendono solo dal valore di a, poichè M è noto, e usa l'algoritmo di Coppersmith per cercare di ricavare k. Il numero primo p (q, rispettivamente) è trovato per il giusto valore di a (b).

Il problema dato è risolto in tre passi:

$$problema \rightarrow f(x) \equiv 0 \bmod p \rightarrow g(x) = 0 \rightarrow x_0$$

Per prima cosa bisogna trasformare il problema dato in una equazione polinomiale modulare $f(x) \equiv 0 \bmod p$, con p ignoto (p divisore di N) e x_0 radice piccola ($f(x_0) \equiv 0 \bmod p$) incognita. La radice x_0 deve essere delimitata superiormente da una sufficientemente piccola costante X, $|x_0| < X$.
Nel secondo passo, l'algoritmo di Coppersmith elimina la variabile p dall'equazione trasformandola in una equazione $g(x) = 0$ a coefficenti interi, tale che g(x) e f(x) abbiano le stesse radici.
Come ultimo passo, il metodo ricava tutte le radici possibili x_0 del polinomio g(x).

Il polinomio g(x) è costruito come una combinazione lineare $g(x) = \sum_i a_i \cdot f_i(x)$ con $a_i \in Z$ e $f_i(x)$ polinomi derivati da f(x). I polinomi $f_i(x)$ sono scelti in modo che abbiano le stessere radici di f(x).
Il polinomio g(x) è ricavato usando l'algoritmo LLL. In sintesi, quello che l'algoritmo LLL produce è una riduzione della base *lattice* $b_o, b_1, ..., b_{n-1}$.

L'algoritmo LLL calcola una base alternativa $b'_o, b_1', ..., b_n - 1'$ *lattice* tale che la nuova base vettoriale sia più piccola della base originale.
Coppersmith usa l'algoritmo LLL per trovare il polinomio g(x) della seguente forma: $g(x) = \sum_{i=0}^{n-1} \cdot b'_{o,i} \cdot x^i$.

L'algoritmo di Coppersmith necessita di sei parametri:

- $f(x)$: polinomio equivalente del problema
- N: modulo pubblico
- β: il limite inferiore di p, tale che $p >= N^\beta$, con $0 < \beta <= 1$.
  In questo paper[1], i ricercatori usano $\beta = 0.5$ poichè la grandezza dei due numeri primi è metà di quella del modulo pubblico N; questo implica che solo uno dei due numeri potrà essere ricavato dall'algoritmo, dato che entrambi i numeri primi non possono essere maggiori di $N^{0.5}$. Ciò non è un problema, dato che ricavato uno dei due numeri primi, per esempio p, l'altro è presto ricavato calcolando $\frac{N}{p} = q$.
- X: limite superiore della piccola radice x_0. In altre parole X, rappresenta il limite superiore di k.
- m e t: variabili che definiscono la matrice B per l'algoritmo LLL

Il costo del metodo è dato dal numero di tentativi di a (ordine del gruppo) e dalla complessità dell'algoritmo di Coppersmith. Il termine ord rappresenta l'ordine moltiplicativo di 65537 nel gruppo Z_M^* che può essere facilmente calcolato.
Nella pratica, ord determina il tempo di esecuzione dell'intera fattorizzazione. Il numero di iterazioni di a è troppo grande anche per chiavi molto piccole. Diminuire il numero di tentativi è necessario per rendere il metodo pratico.

4.3.2 Diminuzione del range di *a*

La prima ottimizzazione che si può implementare è la diminuzione dello spazio di ricerca di a.
Infatti, iterare su tutti i posibili valori di a è impensabile; ad esempio, i valori possibili di a per una chiave di 512 bits sono 2^{62}.

Partiamo riscrivendo (3.2):

$$\begin{aligned} N &= p_1 * p_2 \\ &= (k_1 M + (65537^{a_1} \bmod M)) * (k_2 M + (65537^{a_2} \bmod M)) \\ &= k_1 k_2 M^2 + k_1 M(65537^{a_2} \bmod M) + k_2 M(65537^{a_1} \bmod M) + (65537^{a_2} e^{a_1} \bmod M) \\ &= k_3 * M + (65537^{a_3} \bmod M) \end{aligned}$$

Con $k_3 = k_1 k_2 M + ((k_1 65537^{a_2} + k_2 65537^{a_1}) \bmod M)$ e $a_3 = a_1 + a_2$.
Questo implica che l'esponente pubblico N ha la stessa struttura matematica dei

primi p e q.
Abbiamo visto come a_3 sia facilmente ricavabile da N. Al posto che cercare a nel intervallo $[0, a_3]$, possiamo ridurre l'intervallo a $[\frac{a_3}{2}, \frac{a_3+|G|}{2}]$. Notiamo che $a_1 = a_2 = \frac{a_3}{2}$ è la più piccola combinazione che produce a_3 e $a_1 = a_2 = \frac{a_3+|G|}{2}$ è la più grande. Per tutte le altre combinazioni, solo uno tra a_1, a_2 sarà nell' intervallo, ma questo non incide poichè il nostro metodo può recuperare qualsiasi numero fattore primo di N se indoviniamo il giusto valore di a_1 o a_2 correttamente.

4.3.3 Ulteriori miglioramenti del metodo

Nonostante nel progetto abbinato a questa tesi ci si sia fermati soltanto alla riduzione dell'intervallo di a, esiste un'altra ottimizzazione implementabile. E' stato notato che l'algoritmo di Coppersmith è utilizzabile solo se $\log_2(M) > \log_2(N)/4$. La condizione viene rispettata con un ampio margine, cioè M è molto più grande di quanto sia necessario.
L'idea principale dell'ottimizzazione è introdurre un numero M' con corrispondente riduzione di tentativi $ord_{M'}65537$ tale che i numeri primi abbiano comunque la forma 3.1, con M sostituito da M' e a,k sostituiti da a',k'. Possiamo allora introdurre un nuovo valore M', che sostituisca M, tale che:

1. I fattori p e q abbiano la stessa forma già vista (M' deve essere un divisore di M)
2. $\log_2(M') > \log_2(N)/4$ in modo che la condizione dell'algoritmo sia rispettata
3. Il tempo di esecuzione del metodo sia minima (numero di tentativi e tempo per tentativo)

I valori ottimizzati di M' per differenti lunghezze di chiave sono stati trovati insieme ai valori m e t attuando una ricerca locale attraverso *brute force* ottimizzata da un algoritmo *greedy*. Lo studio approfondito sull'ottimizzazione dei valori è visionabile in questo articolo[1].

Algorithm 1: Fattorizzazione di chiavi pubbliche RSA generate da RSALib

input : N,M',m,t
output: p-fattore di N
c' $\leftarrow \log_{65537} N \bmod M'$; ▷ `Usa Pohlig-Hellman alg.`
ord' $\leftarrow$ ord M' (65537) **forall** $a \in [a_3/2, \frac{a_3+|G|}{2}]$ **do**
  f(x) $\leftarrow x + (M'^{-1} \bmod N) \bullet (65537^{a'} \bmod M')(\bmod N)$ $(\beta, X) \leftarrow (0.5, 2$ $N^{\beta}/M')$; ▷ `Impostazione dei parametri`
  k' $\leftarrow$ Coppersmith(f(x), N, β, m, t, X) p $\leftarrow k' \bullet M' + (65537^{a'} \bmod M')$; ▷ `Candidato fattore`
  if $N \bmod p = 0$ **then**
    return p
  end
end

Appendice A

Implementazione del programma

Una pratica implementazione dell'attacco è stata programmata su Phyton 3.8.6 e Sagemath 9.1.
Le librerie esterne usate sono sympy, Cryptodome, math, labmath.
Il programma è fruibile a questo indirizzo [5].

A.1 Generazione delle chiavi

La costruzione dei primi p e q della forma 3.1 è stata possibile definendo la funzione *generator_of_primes*.
La funzione ha come input M, il primoriale, la lunghezza in bits di a, k, e la lunghezza desiderata della chiave; *a_bits*, *k_bits* e M sono calcolati esternamente a questa funzione.

```
def generator_of_primes(a_bits,k_bits,M,bits_key):
#Genera i primi p e q basati sulla formula
#Input:a_bits,k_bits,M,bits_key
#Output:i numeri primi p,q e il loro prodotto N
    N=0
    while N.bit_length()!=bits_key:

        while True:
            a_p=getRandomNBitInteger(a_bits)
            k_p=getRandomNBitInteger(k_bits)
            p=k_p*M+pow(e,a_p,M)
            if isprime(p):
                break

        while True:
            a_q=getRandomNBitInteger(a_bits)
            k_q=getRandomNBitInteger(k_bits)
            q=k_q*M+pow(e,a_q,M)
            if isprime(q):
                break
        N=p*q
    return N,p,q
```

Il procedimento è semplice:

- sia per p che per q vengono generati valori casuali di a e k della lunghezza necessaria, attraverso il richiamo della funzione *getRandomNBitInteger* dalla libreria *Cryptodome.Util.number*.
- Si generano p e q secondo la struttura 3.1 e si controlla la loro primalità; nel caso non siano entrambi numeri primi si ripete da capo.
- Si calcola $N = p \cdot q$ e si controlla che N sia della lunghezza desiderata (*bits_key*); nel caso la lunghezza non sia rispettata si riparte da capo.

A.2 Fingerprinting

Per il processo di fingerprinting si è definita la funzione *fingprint*, che accetta come unico input il modulo pubblico N. Calcolando la lunghezza del modulo si può ottenere il primoriale associato, richiamando la funzione *bits_check* e la funzione *primorial* della libreria sympy.
In seguito si tenta di calcolare il logaritmo discreto $c = log_{65537}N \bmod M$ usando la funzione *_discrete_log_pohlig_hellman* dalla libreria *sympy.ntheory.residue_ntheory*. Nel caso l'esecuzione della funzione *_discrete_log_pohlig_hellman* andasse a buon fine, ritornerebbe True, nel caso non riuscisse a trovare una soluzione al logaritmo, allora ritornerebbe False.

```
def fingprint(N):
#Fingerprinting della chiave fornita 'N'.
#Input: N, chiave pubblica
#Output: True or False, True=Chiave vulnerabile

    #Calcola il primoriale della chiave e con il logaritmo discreto di
    #pohlig-hellman calcolo l'esponente;
    #Se la funzione _discrete_log_pohlig_hellman(M,N,e) va a buon fine
    #la chiave    vulnerabile e ritorniamo True
    n=bits_check(N.bit_length())
    M=primorial(n)
    try:
        c=_discrete_log_pohlig_hellman(M,N,e)
        return True
    except:
        return False
```

A.3 Implementazione attacco

Per l'attacco viene definita una funzione *solve* che ha come input tutti i parametri dell'attacco e che ritorna il valore di p e q se è stato possibile ricavarlo.
Notare come a, la parte nota dell'algoritmo, faccia parte degli input; infatti la funzione solve viene richiamata all'interno di un ciclo for che itera il valore di a all'interno dell'intervallo ottimizzato calcolato.

```
def solve(M, n, a, m, t, beta):
#Crea il polinomio monico associato al problema
#Input:M,N,a
#Output:p e q, i due numeri primi che fattorizzano N

    #Calcola la parte nota di p: 65537^a * M^-1 (mod N)
    p_noto = int(pow(e, a, M) * inverse_mod(M, n))

    #Crea il polinomio f(x)
    A = PolynomialRing(Zmod(n), implementation='NTL', names=('x',))
    (f,) = A._first_ngens(1)
    polinomio = f + p_noto

    #Limite superiore della radice piccola x0
    Lim_sup = floor(2 * n**0.5 / M)

    #Trova la radice piccola usando l'algoritmo di Coppersmith
    radici = coppersmith_howgrave_univariate(polinomio, n, beta, m, t,
   Lim_sup)

    #Se a    sbagliato non esisteranno radici
    for i in radici:

        #Ricostruisce P dalla k calcolata
        p = int(i*M + pow(e, a, M))
        if n%p == 0:
            return [p, n//p]
```

La fasi della funzione solve sono :

- Vengono calcolati i bits noti attraverso $pow(e, a, M) * inverse\_mod(M, n)$
- Viene creato il polinomio monico modulare grazie alla funzionalità di sage *PolynomialRing*
- Viene calcolato il limite superiore della radice piccola x_0
- Viene richiamata la funzione *coppersmith_howgrave_univariate* che attua l'algoritmo di Coppersmith-Howgrave. Questa funzione ritorna le radici del polinimio inserito in input. Nel caso i valori di input della funzione siano sbagliati, non ritorna nessuna radice.
- Vengono ciclate le radici trovate e vengono costruiti i possibili primi p. Se $\frac{N}{p}$ ha resto nullo allora p è il primo che fattorizza N e vengono ritornati p e $q = \frac{N}{p}$.

La funzione *coppersmith_howgrave_univariate* è tratta dalla seguente fonte [6].

Bibliografia

[1] M. Nemec, M. Sys, P. Svenda, D. Klinec, and V. Matyas, *The Return of Coppersmith's Attack:Practical Factorization of Widely Used RSA Moduli*. ACM CCS, 2017.

[2] "Schema crittografia asimmetrica." https://commons.wikimedia.org/wiki/File:Crittografia_asimmetrica_schema.png.

[3] "How does the roca attack work?." https://crypto.stackexchange.com/questions/53906/how-does-the-roca-attack-work.

[4] "Analysis of the roca vulnerability." https://bitsdeep.com/posts/analysis-of-the-roca-vulnerability/.

[5] P. Torasso, "Attacco-roca-sulla-vulnerabilita-cve-2017-15361." https://github.com/Elbarbons/Attacco-ROCA-sulla-vulnerabilita-CVE-2017-15361, 2020.

[6] "Rsa-and-lll-attacks." https://github.com/mimoo/RSA-and-LLL-attacks/blob/master/coppersmith.sage.