Estrazione di numeri primi da un flusso di bit casuali

Mattia Tarabolo

Anno accademico: 2019/20

Università degli Studi dell'Insubria

Dipartimento di Scienza ed Alta Tecnologia
Corso di Laurea Triennale in Fisica (L-30)

Estrazione di numeri primi da un flusso di bit casuali

Mattia Tarabolo
Numero di matricola: 733165

Relatore:

Prof. Massimo Caccia

Anno accademico: 2019/20

Mattia Tarabolo
Estrazione di numeri primi da un flusso di bit casuali
Anno accademico: 2019/20
Relatore: Prof. Massimo Caccia

Università degli Studi dell'Insubria
Dipartimento di Scienza ed Alta Tecnologia
Via Valleggio 11
22100 Como

Indice

Elenco delle figure

1 Introduzione

Da sempre la crittografia è stata fondamentale per permettere la comunicazione segreta fra due enti in presenza di terze parti. Già ai tempi degli antichi greci i generali utilizzavano particolari tecniche crittografiche per comunicare strategie segrete evitando il rischio che queste venissero a conoscenza dei nemici in caso di intercettazione del messaggio. Col tempo i metodi crittografici si sono evoluti sempre più, fino alla nascita dei computer. Grazie allo sviluppo di questi potenti strumenti, la crittografia dovette innovarsi sempre più, per costruire nuovi metodi inviolabili da qualsiasi computer. In particolare, con la diffusione dei personal computer, la comunicazione sicura fra due utenti privati tramite la rete internet si è resa necessaria. Le vecchie tecniche crittografiche, che richiedevano la condivisione di una chiave crittografica segreta fra i due utenti comunicanti, era poco applicabile per tale scopo a causa delle difficoltà nella condivisione sicura della chiave. Nacque così un nuovo tipo di schema crittografico, detto crittografia a chiave pubblica, che non richiedeva più la condivisione di una chiave segreta. Fra questi nuovi schemi, il più utilizzato è la crittografia RSA [1] [2], per il cui funzionamento è richiesta la generazione di grandi numeri primi. Tale problema non è affato di semplice implementazione, e richiede uno sforzo computazionale non banale.

In questo lavoro di tesi si presenteranno prima le basi della crittografia, e in particolare si spiegherà in breve il funzionamento della crittografia RSA, con particolare enfasi sull'importanza della generazione di numeri primi. In seguito verrà presentato il problema della generazione di numeri primi grandi, e verranno presentati alcuni algoritmi tradizionali e un algoritmo efficiente per tale scopo. L'algoritmo efficiente, sviluppato seguendo il brevetto [3], verrà poi implementato su un personal computer tramite il linguaggio di programmazione Python, e verrà confrontato con un algoritmo tradizionale.

1.1 Risultati

L'algoritmo di generazione di numeri primi efficiente è implementato efficacemente su un personal computer, assieme a un algoritmo di generazione tradizionale.

Durante il confronto si è costruita una metrica in grado di mostrare quanto l'algoritmo efficiente sia realmente migliore di quello tradizionale. Si vedrà infatti che questo permette di risparmiare un gran numero di operazioni.

2 Crittografia

Fin dall'antichità l'uomo si è servito della *crittografia* (dal greco antico: *kryptós* "nascosto, segreto"; e *graphein* "scrivere"), ovvero della pratica di tecniche che permettano una comunicazione sicura e confidenziale in presenza di terzi, detti *avversari* o *attaccanti*. Più in generale, la crittografia si occupa di sviluppare e analizzare tecniche in grado di assicurare la segretezza di dati privati in presenza di avversari.

Già gli egizi utilizzavano tecniche crittografiche, seppure in maniera limitata, ma tale scienza si sviluppò principalmente sotto la spinta di forze militari e politiche, le quali avevano la necessità di proteggere strategie e segreti e di comunicare in modo sicuro. Uno dei metodi di cifratura più semplici e antichi è il cosiddetto *cifrario di Giulio Cesare*: ogni lettera dell'alfabeto veniva sostituita con quella che si trovava alla sua destra di 3 posizioni, rendendo così il messaggio incomprensibile a un eventuale intercettatore. Per eseguire correttamente la cifratura, l'alfabeto è posto in una struttura ad anello in cui alla lettera z segue la lettera a. La sicurezza del metodo si basa sull'assunzione che solo il mittente e il ricevente conoscono il processo di codifica [2].

Negli anni si sono sviluppati nuovi metodi sempre più avanzati, ma la maggiore spinta innovativa deriva dall'invenzione e dalla diffusione dei computer e dei sistemi di comunicazione a distanza. Gli obiettivi della crittografia si ampliarono, e alla *confidenzialità* di un messaggio si aggiunsero l'*integrità di dati* e l'*autenticazione* [4].

2.1 Terminologia

Nell'ambito della confidenzialità si parla di *cifratura*, o anche *codifica*, quando un messaggio confidenziale viene modificato attraverso un algoritmo, detto *algoritmo di cifratura*, in modo da renderlo completamente incomprensibile a un avversario; il processo inverso che permette di passare dal messaggio codificato al messaggio originale viene invece detto *decodifica*. Solitamente il messaggio codificato viene spedito dal *mittente* (in letteratura spesso indicato come Alice, o A) al *ricevente*

(in letteratura Bob, o B). Se solo A e B conoscono l'algoritmo di codifica potranno comunicare in modo sicuro, anche se il messaggio cifrato viene intercettato da un avversario.

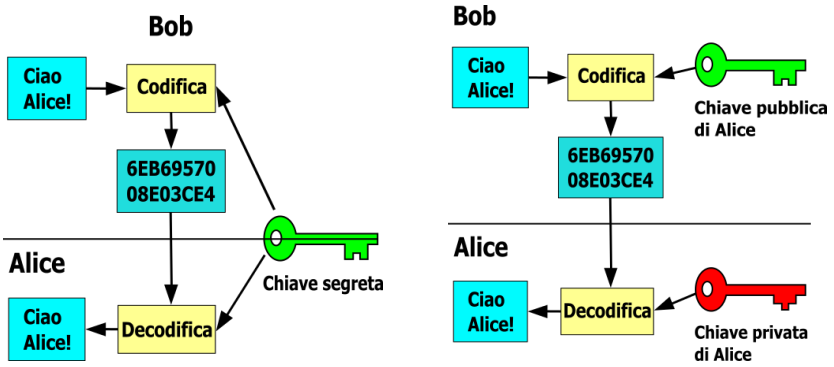
Normalmente gli algoritmi di cifratura sono basati su una *chiave crittografica*, la quale deve essere rigorosamente mantenuta segreta e condivisa solo fra A e B. In questo modo un eventuale avversario, anche conoscendo il particolare algoritmo di codifica usato, dovrebbe venire a conoscenza anche della chiave crittografica per poter decifrare i messaggi.

Con l'avvento dei computer gli attacchi ai sistemi crittografici sono stati resi sempre più semplici, e per questo motivo si sono dovuto sviluppare algoritmi sempre più efficaci. Al giorno d'oggi non esistono algoritmi che siano teoreticamente inviolabili, ma solo algoritmi *computazionalmente inviolabili*. Questo significa che il tempo necessario a ricavare informazioni sulla chiave crittografica al fine da violare il sistema è talmente elevato da rendere l'attacco inefficace.

2.2 Schemi di codifica

Finora abbiamo sempre descritto uno schema di codifica in cui entrambe le parti coinvolte nella comunicazione sono a conoscenza della chiave crittografica. Tale tipo di schema viene detto *crittografia simmetrica*, e si basa sull'utilizzo di una funzione di codifica invertibile che trasforma il messaggio originale in quello cifrato attraverso l'utilizzo della chiave, e viceversa. Si riporta uno schema crittografico simmetrico in figura 2.1a. Il metodo più utilizzato al momento è il cosiddetto *Advanced Encryption Standard* (AES), usato come standard dal governo degli Stati Uniti d'America. Tale schema può essere violato facilmente se la chiave crittografica non è condivisa in modo sicuro, rendendolo quindi poco efficiente nell'uso privato.

Oltre allo schema simmetrico, ricopre un ruolo di fondamentale importanza la *crittografia asimmetrica*, o anche *crittografia a chiave pubblica*. Questa coinvolge l'uso di due diverse chiave crittografiche generate dal ricevente A, di cui una è mantenuta privata (detta *chiave privata*) e una è distribuita al pubblico (detta *chiave pubblica*). Il mittente B può così cifrare il messaggio attraverso una funzione non invertibile e la chiave pubblica, per poi spedirlo a A che è in grado di decifrarlo attraverso l'uso di un'altra funzione e della chiave privata. Il metodo più utilizzato al momento è la *crittografia RSA*. Tale schema si fonda sull'assunzione che la conoscenza della chiave pubblica non permetta in alcun modo di ricavare la chiave privata. Si riporta uno schema crittografico asimmetrico in figura 2.1b.



(a) Crittografia simmetrica **(b)** Crittografia asimmetrica

Fig. 2.1.: Schemi di crittografia simmetrica (a) e asimmetrica (b).

2.3 Crittografia RSA

Per ovviare al problema della distribuzione delle chiavi in un sistema crittografico simmetrico, Diffie e Hellman presentarono in un rivoluzionario articolo del 1976 [5] il primo algoritmo crittografico asimmetrico. Solo due anni dopo Rivest, Shamir e Adleman svilupparono al Massachusetts Institute of Technology (MIT) il cosiddetto *schema RSA*, dalle iniziali dei loro cognomi [1]. Da quel momento il metodo è stato il più usato nel campo della crittografia a chiave pubblica.

In generale tutti gli schemi a chiave pubblica sono basati su una profonda conoscenza della *teoria dei numeri*, ovvero quel ramo della matematica che si occupa dello studio delle proprietà dei numeri interi.

2.3.1 Concetti matematici

Prima di presentare l'algoritmo crittografico, verranno introdotte alcune definizioni e proposizioni necessari a comprendere la matematica sottostante. Queste saranno utili anche più avanti, nello studio della generazione di numeri primi.

Teoria dei numeri

D'ora in poi l'insieme dei numeri interi $\{\ldots, -2, -2, 0, 1, 2, \ldots\}$ sarà indicato dal simbolo $\mathbb{Z}$. Se un intero a *divide* un intero b (o equivalentemente a è *divisore* o *fattore* di b) si indica $a|b$.

Se $a, b \in \mathbb{Z}$ con $b \geq 1$, allora la *divisione euclidea* di a con b produrrà due interi q (detto *quoziente*) e r (detto *resto*) tali che $a = qb + r$ e $0 \leq r < b$. In questo modo è possibile definire l'*operazione modulo* fra i due interi a e b, indicata come $a \mod b$, che restituisce il resto della loro divisione euclidea. L'operazione modulo gode delle seguenti proprietà sotto l'azione delle operazioni di somma, sottrazione e prodotto fra interi:

$$(a \pm b) \mod n = [(a \mod n) \pm (b \mod n)] \mod n \tag{2.1}$$

$$ab \mod n = [(a \mod n)(b \mod n)] \mod n \tag{2.2}$$

Due interi a e b sono detti *relativamente primi* o *coprimi* se il *massimo comune divisore* dei due interi è 1, ovvero $\text{MCD}(a, b) = 1$. Equivalentemente, a e b non hanno divisori comuni se non 1.

Un intero $p \geq 2$ è detto *numero primo* se gli unici suoi divisori sono 1 e p stesso.
Il *teorema fondamentale dell'aritmetica* afferma che un qualsiasi intero $n \geq 2$ può sempre essere fattorizzato come prodotto di potenze di numeri primi $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$, dove p_i sono numeri primi distinti e e_i interi.

Dato un qualsiasi intero $n \geq 1$ si può definire la *funzione φ di Eulero* o semplicemente *funzione di Eulero* o *toziente* $\varphi(n)$ come il numero di interi coprimi a n nell'intervallo $[1, n]$. Si può mostrare che dato un numero primo p vale $\varphi(p) = p - 1$. Inoltre si dimostra che la funzione di Eulero è *moltiplicativa*, ovvero se m è un intero coprimo a n allora $\varphi(mn) = \varphi(m)\varphi(n)$. Da queste due proposizioni assieme al teorema fondamentale dell'aritmetica segue facilmente che data la fattorizzazione in numeri primi di n la sua funzione di Eulero è:

$$\varphi(n) = n \left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_k}\right) \tag{2.3}$$

Esempio:
Sia $n = 9$, i numeri coprimi a 9 nell'intervallo $[1, 8]$ sono 1, 2, 4, 5, 7, 8. Segue allora $\varphi(9) = 6$ dalla definizione. Inoltre fattorizzando si ha $9 = 3^2$, da cui $\varphi = 9(1 - 1/3) = 6$.

Aritmetica modulare

L'operazione modulo permette di definire una *relazione di equivalenza* fra gli interi. In particolare dato un intero $n \geq 1$ e $a, b \in \mathbb{Z}$ si dice che a è *congruente* a b *modulo* n,

e si indica $a \equiv b \pmod{n}$, se n divide $(a - b)$ o equivalentemente se $a \mod n = b \mod n$. In questo modo sono definite esattamente n classi di equivalenza per ogni intero n, dette anche *classi di resto modulo* n.

L'insieme delle classi di equivalenza modulo n, assieme all'addizione e moltiplicazione, formano un *anello* detto *anello degli interi modulo* n. Questo è indicato dai simboli $\mathbb{Z}/n\mathbb{Z}$ o $\mathbb{Z}_n$. Si ha allora:

$$\mathbb{Z}_n = \{[0], [1], ..., [n-1]\} \tag{2.4}$$

dove $[r] = \{a \in \mathbb{Z} \,|\, a \equiv r \pmod{n}\}$ è la classe di resto r modulo n. le operazioni di addizione, sottrazione e moltiplicazione in $\mathbb{Z}_n$ sono definite come:

$$[a] \pm [b] = [a \pm b] \tag{2.5}$$

$$[a][b] = [ab] \tag{2.6}$$

Dato $a \in \mathbb{Z}_n$, si definisce l'*inverso moltiplicativo di* a *modulo* n l'intero $x \in \mathbb{Z}_n$ tale che $ax \equiv 1 \pmod{n}$. Se x esiste allora si può mostrare che è anche unico, e a è detto *invertibile* o *unità*. Si può dimostrare che a è invertibile se e solo se $\text{MCD}(a, n) = 1$, e dunque a è coprimo a n.

> **Esempio:**
> Sia $n = 9$, gli elementi invertibili in $\mathbb{Z}_9$ sono 1, 2, 4, 5, 7, 8. Per esempio $4 \cdot 7 \equiv 1 \pmod{9}$, o anche $\text{MCD}(4, 9) = \text{MCD}(7, 9) = 1$.

Si può infine definire il *gruppo moltiplicativo* di $\mathbb{Z}_n$ come il gruppo delle sue unità. Questo viene indicato con il simbolo $\mathbb{Z}_n^*$, ed è dunque il gruppo degli elementi di $\mathbb{Z}_n$ coprimi a n:

$$\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \,|\, \text{MCD}(a, n) = 1\} \tag{2.7}$$

Poiché tale gruppo è chiuso sotto l'azione della moltiplicazione, segue che se $a, b \in \mathbb{Z}_n^*$ allora $a \cdot b \in \mathbb{Z}_n^*$.

> **Esempio:**
> Sia $n = 9$, per quanto visto finora $\mathbb{Z}_9^* = \{1, 2, 4, 5, 7, 8\}$.

2.3.2 Descrizione dell'algoritmo

Lo schema crittografico RSA fa uso di due chiavi crittografiche, ognuna composta da una coppia di interi: la chiave privata (n, e) e la chiave pubblica (n, d) [2].

Il messaggio viene suddiviso in blocchi, ognuno dei quali viene poi trasformato in un intero positivo $m < n$ (per esempio passando prima alla notazione binaria, e poi trasformando in un intero).
Ogni blocco m viene allora cifrato dal mittente B in un intero c usando la chiave pubblica tramite l'operazione:

$$c = m^e \mod n \tag{2.8}$$

Il ricevente A decodifica dunque il messaggio cifrato, utilizzando la chiave privata tramite l'operazione:

$$m = c^d \mod n = (m^e \mod n)^d \mod n = m^{ed} \mod n \tag{2.9}$$

L'algoritmo crittografico funziona se sono soddisfatte le seguenti richieste:

1. É possibile calcolare e, d e n tali che $m^{ed} \mod n = m$ per ogni $m < n$.
2. É relativamente semplice calcolare le relazioni 2.8 e 2.9.
3. É infattibile determinare d conoscendo esclusivamente e e n.

Si può mostrare che la 1 è verificata se e e d sono inversi moltiplicativi modulo $\varphi(n)$, ovvero se:

$$ed \equiv 1 \pmod{\varphi(n)} \tag{2.10}$$

É possibile trovare la dimostrazione in appendice R di [2].
Questo significa che e e d sono coprimi a $\varphi(n)$, ovvero $e, d \in \mathbb{Z}^*_{\varphi(n)}$. Le altre due richieste sono invece soddisfatte scegliendo n come il prodotto di due numeri primi p e q, ovvero $n = pq$. In questo modo si trova facilmente che vale $\varphi(n) = (p-1)(q-1)$.

Esempio:
Si scelgono due numeri primi $p = 17$ e $q = 11$, da cui $n = 17 \cdot 11 = 187$. In questo modo si ha $\varphi(n) = 16 \cdot 10 = 160$. Si sceglie allora un intero $e < \varphi(n) = 160$ coprimo a $\varphi(n) = 160$, per esempio $e = 7$, e si determina il suo inverso moltiplicativo $d \in \mathbb{Z}_{160}$ tale che $d \cdot 7 \equiv 1 \pmod{160}$. Nel nostro caso $d = 23$ perché $23 \cdot 7 \equiv 1 \pmod{160}$.
Consideriamo ora un messaggio $m = 88$, questo viene cifrato in $c = 88^7 \mod 187 = 11$, che può essere calcolato usando le proprietà moltiplicative dell'operazione modulo. Il messaggio cifrato viene poi decodificato, usando $m = 11^{23} \mod 187 = 88$.

Lo schema RSA può essere allora riassunto in tre fasi: generazione delle chiavi da parte del ricevente A; cifratura del messaggio da parte del mittente B tramite la chiave pubblica; decodifica del messaggio da parte del ricevente A tramite la chiave privata.

Algoritmo 1. Generazione delle chiavi da parte del ricevente A

1: Si selezionano due numeri primi p e q differenti
2: Si calcola $n \leftarrow pq$
3: Si calcola la funzione di Eulero $\varphi(n) \leftarrow (p-1)(q-1)$
4: Si sceglie un intero $e \in \mathbb{Z}^*_{\varphi(n)}$
5: Si calcola $d \in \mathbb{Z}^*_{\varphi(n)}$ tale che $ed \equiv 1 \pmod{\varphi(n)}$

Algoritmo 2. Cifratura di un messaggio da parte del mittente B

1: Si separa il messaggio in blocchi e ogni blocco viene trasformato in un intero $m < n$
2: Ogni intero viene cifrato in $c \leftarrow m^e \mod n$

Algoritmo 3. Cifratura di un messaggio da parte del mittente B

1: Ogni intero cifrato viene decodificato in $m \leftarrow c^d \mod n$
2: Il messaggio originale è ricostruito assemblando i diversi blocchi e ritrasformando nel linguaggio originale

2.3.3 Aspetti computazionali

Nell'implementazione di uno schema crittografico ricopre un ruolo di fondamentale importanza la complessità computazionale. Se infatti un algoritmo risulta troppo complesso, questo non sarà applicabile in quanto il tempo necessario per svolgere i calcoli sarebbe troppo grande, rendendolo quindi inefficiente.

Nel caso della crittografia RSA un punto critico è il calcolo dell'elevamento a potenza modulare, ovvero di una operazione del tipo $a^e \mod n$. Quando sia a che e sono grandi, svolgere prima l'elevamento a potenza per poi ridurre modulo n porterebbe ad avere un valore intermedio gigantesco, difficilmente gestibile da un qualsiasi processore. Tuttavia utilizzando le proprietà dell'operazione modulo il calcolo può essere svolto in maniera efficace e veloce.

Si riscrive l'esponente e in forma binaria:

$$e = \sum_{i=0}^{k-1} e_i 2^i \tag{2.11}$$

dove $e_i = 0, 1$ e $e_{k-1} = 1$ per definizione. In questo modo l'elevamento a potenza può essere ridotto alla forma:

$$a^e \mod n = a^{\sum_{i=0}^{k-1} e_i 2^i} \mod n = \left[\prod_{i=0}^{k-1} a^{e_i 2^i}\right] \mod n =$$
$$= \left(\prod_{i=0}^{k-1} \left[a^{e_i 2^i} \mod n\right]\right) \mod n$$

ovvero:

$$a^e \mod n = \left(\prod_{e_i \neq 0} \left[a^{2^i} \mod n\right]\right) \mod n \tag{2.12}$$

Il calcolo si riduce allora all'elevamento a potenza modulare dei soli termini con $e_i \neq 0$, e la complessità computazionale è estremamente ridotta. L'algoritmo implementato in Python è riportato in 12.

```
1  def modpow(b, e, m):
2    if m == 1:
3      return 0
4    b = b % m
5    c = 1
6    while e > 0:
7      if (e % 2) == 1:
8        c = (c * b) % m
9      e = e >> 1
10     b = (b * b) % m
11   return c
```

Listing 2.1.: Funzione di elevamento a potenza modulare implementata in Python.

Al posto di passare alla rappresentazione binaria di e per poi controllare quali valori di e_i fossero non nulli, si è usato la proprietà che un numero dispari ha l'ultimo bit uguale a 1 in rappresentazione binaria, assieme all'operazione $e >> 1$ che elimina l'ultimo bit di e.

2.3.4 Sicurezza dello schema RSA

La sicurezza della crittografia RSA si basa sulla complessità computazionale della fattorizzazione. Infatti la fattorizzazione di un intero n composto da numeri primi grandi è un compito molto difficile.

Per questo motivo le chiavi RSA vengono scelte in modo che i numeri primi che le compongono siano più grandi possibili. In particolare si parla di chiave RSA a k-bit

se n è un intero a k-bit (ovvero $2^{k-1}+1 \leq n \leq 2^k-1$). Per assicurare che p e q siano il più grande possibile, questi sono scelti come numeri primi a $k/2$-bit nell'intervallo $\left[\sqrt{2^{2k-1}+1}, 2^k-1\right]$.

Al fine di tracciare le nuove scoperte nel campo della fattorizzazione, in modo da poter determinare quale grandezza delle chiavi RSA sia considerabile sicura, il 18 marzo 1991 gli *RSA Laboratories* indirono un concorso a premi dal nome *RSA Factoring Challenge* [2]. Il concorso offriva un premio in denaro a chiunque fosse riuscito a fattorizzare alcune chiavi RSA di diversa lunghezza.
La prima a essere fattorizzata fu una chiave a 330-bit nell'aprile del 1991 da parte di Lenstra [@6]. La fattorizzazione venne effettuata in pochi giorni usando un algoritmo chiamato *crivello quadratico* (o *quadratic sieve* in inglese, abbreviato in *QS*) su un MasPar computer. Usando invece moderni microprocessori la fattorizzazione avviene in qualche ora.
Nell'aprile del 1996 invece un team diretto sempre da Lenstra fattorizzò per la prima volta una chiave a 431-bit, usando un nuovo algoritmo chiamato *crivello dei campi di numeri generale* (o *general number field sieve* in inglese, abbreviato in *GNFS*) [@6]. Tale algoritmo è tuttora il più potente e il più usato: grazie al miglioramento della tecnologia e della potenza di calcolo dei computer, assieme a un miglioramento dell'algoritmo stesso, a febbraio 2020 venne fattorizzata una chiave RSA a 829-bit (la più grande finora fattorizzata) [@7].

La crittografia RSA è allora sicura solo se le chiavi sono scelte sufficientemente grandi. Al giorno d'oggi è consigliato utilizzare chiavi a 2048-bit, in quanto si stima che nell'ultimo decennio sarà possibile fattorizzare le ormai obsolete chiavi a 1024-bit.

Nell'applicazione dello schema crittografico risulta allora fondamentale lo sviluppo di un algoritmo di generazione di numeri primi grandi. Come si vedrà nel prossimo capitolo questo problema non è affatto banale, e richiede uno sforzo computazionale elevato se non trattato in modo efficiente.

Oltre alla dimensione di n, per rendere lo schema ancora più sicuro è importante che i due numeri primi p e q siano casuali e distribuiti uniformemente nell'intervallo $\left[\sqrt{2^{2k-1}+1}, 2^n-1\right]$. Se infatti questi non fossero distribuiti uniformemente, sarebbe più facile trovare la fattorizzazione di n partendo prima dalla combinazione di primi più probabile, e testando tutte le altre combinazioni in ordine decrescente di probabilità, fino a trovare i valori corretti.

Bisogna infine tenere conto degli enormi progressi nel campo della computazione quantistica. Nel 1994 Shor propose l'omonimo algoritmo di fattorizzazione quanti-

stico, che permetterebbe di fattorizzare un qualsiasi intero in tempi estremamente ridotti rispetto a un algoritmo classico [8]. Il limite più grosso alla sua implementazione è tuttavia la costruzione di computer quantistici sufficientemente potenti: se in futuro questo sarà possibile, la crittografia RSA dovrà essere abbandonata, in quanto verrà a mancare la sicurezza basata sulla complessità computazionale della fattorizzazione.

2.3.5 Conclusioni

Si è visto come nella costruzione di uno schema RSA ricopra un ruolo fondamentale la generazione di numeri primi uniformemente distribuiti in un intervallo sufficientemente grande. In particolare l'algoritmo di generazione deve essere sufficientemente efficiente, in modo da poter generare tali numeri in un tempo relativamente breve.

Come si vedrà nel capitolo seguente il compito richiede uno sforzo computazionale non banale se trattato tradizionalmente. Si mostrerà tuttavia come con una serie di accorgimenti i tempi di generazione possano essere estremamente ridotti.

3 Generazione di numeri primi

La generazione di numeri primi risulta di fondamentale importanza nella crittografia, come visto nel capitolo precedente. Lo studio dei numeri primi è inoltre di fondamentale interesse nella teoria dei numeri, tanto che ne sono stati scritti interi libri al riguardo [9][10].

Nella sezione 2.3.1 si è già definito quando un intero è un numero primo, e si è enunciato il teorema fondamentale dell'algebra. Per comprendere meglio gli algoritmi che verranno presentati è tuttavia necessario introdurre ulteriori definizioni e proposizioni.

3.1 Fondamenti matematici

Nel campo della teoria dei numeri il *teorema dei numeri primi* è di fondamentale importanza. Il teorema afferma che, sia $\pi(x)$ il numero di numeri primi minori di un intero x, allora vale:

$$\lim_{x\to\infty} \frac{\pi(x)}{x/\ln x} = 1 \tag{3.1}$$

In particolare si può mostrare che vale:

$$\pi(x) = \mathrm{Li}(x) + O\left(xe^{-\sqrt{\ln x}/15}\right) \qquad \text{per} \qquad x \to \infty \tag{3.2}$$

dove $\mathrm{Li}(x)$ è la funzione logaritmo integrale definita come $\mathrm{Li}(x) = \int_2^x 1/(\ln y)\,dy$. Sviluppando tale funzione si trova il risultato del teorema.

> **Esempio:**
> Sia $x = 10^{10}$, il numero di numeri primi minori di x è $\pi(x) = 455\,052\,511$ mentre $x/\ln x = 434\,294\,481$.

Nel campo dell'aritmetica modulare è invece di fondamentale importanza il *piccolo teorema di Fermat*, che afferma che se p è un numero primo e $a \in \mathbb{Z}_p$, ovvero a non è divisibile da p, allora è verificata l'equivalenza:

$$a^{p-1} \equiv 1 \pmod{p} \tag{3.3}$$

É possibile trovarne una dimostrazione in [2] a pagina 64.

Esempio:
Sia $p = 5$, allora $\mathbb{Z}_m^* = \{1, 2, 3, 4\}$. Si ha dunque:

$$1^4 = 1 \equiv 1 \pmod 5 \qquad 2^4 = 16 \equiv 1 \pmod 5$$
$$3^4 = 81 \equiv 1 \pmod 5 \qquad 4^4 = 256 \equiv 1 \pmod 5$$

Infine enunciamo un teorema fondamentale per quanto sarà mostrato in seguito, detto *teorema di Carmichael*. Questo afferma che se a e m sono due interi positivi coprimi fra loro, vale la congruenza:

$$a^{\lambda(m)} \equiv 1 \pmod m \tag{3.4}$$

dove $\lambda(m)$ è la cosiddetta *funzione di Carmichael*. Sia $m = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ la fattorizzazione in numeri primi di m, allora tale funzione è definita come:

$$\lambda(m) = \text{MCM}\left(\lambda(p_1^{e_1}), \lambda(p_2^{e_2}), ..., \lambda(p_k^{e_k})\right) \tag{3.5}$$

e:

$$\lambda(p^e) = \begin{cases} \varphi(p^e) & \text{se } p \text{ dispari o } p^e = 2, 4 \\ \frac{1}{2}\varphi(p^e) & \text{altrimenti} \end{cases} \tag{3.6}$$

É possibile trovarne una dimostrazione in [11] a pagina 39.

3.2 Il problema della generazione di numeri primi

Ad oggi non esiste alcuna formula analitica che permetta di generare un numero primo. Questi infatti sembrano essere distribuiti casualmente nell'insieme dei numeri interi, e non esiste una particolare relazione che li leghi.

Se si vogliono calcolare tutti i numeri primi più piccoli di un certo intero N, con N relativamente piccolo, si può usare il metodo del *crivello di Eratostene* [10]. Si procede nel seguente modo:

1. Si scrivono tutti gli interi positivi da 2 a N.
2. Si cancellano tutti i multipli di 2 maggiori di 2.
3. In ogni passaggio successivo si cancellano tutti i multipli del più piccolo numero rimasto p maggiori di p.

4. Si ripete il processo per ogni $p < \sqrt{N}$.

I numeri rimasti sono dunque numeri primi.

Esempio:
Sia $N = 101$, si ha allora:

	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~	~~50~~
~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	~~77~~	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89	~~90~~
~~91~~	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~	~~100~~
101									

e dunque i numeri primi minori di 101 sono 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101.

Se si vogliono generare numeri primi grandi, come serve per esempio nella crittografia RSA, questo metodo è estremamente inefficace. Nessun calcolatore sarebbe infatti in grado di eseguire tale algoritmo in un tempo sufficientemente breve.
Bisogna in tal caso avvalersi di altri metodi, che sono tutti basati sull'estrazione di un candidato in un certo sottoinsieme dei numeri positivi. Tale candidato viene poi sottoposto a dei *test di primalità*, che verificano se sia o meno un numero primo. La ricerca avviene fino a quando non si trova un numero primo [4].

3.3 Test di primalità

Come visto nella sezione 3.2 i test di primalità sono fondamentali al fine della generazione di grandi numeri primi. Questi si dividono in due grandi categorie.

Test di primalità deterministici Determinano con certezza se un numero è primo. Qualsiasi numero composto è identificato come tale.

Test di primalità probabilistici Determinano solo con una certa probabilità se un numero è primo. Tale numero viene dunque detto *probabilmente primo* o *primo probabile* o più semplicemente *pseudoprimo*. Esistono numeri composti che vengono identificati come primi.

Fino al 2002 non esisteva nessun tipo di test deterministico non basato su congetture non dimostrate. Tutti quelli usati in precedenza si basavano infatti su alcune congetture non ancora provate: i *test di primalità a curve ellittiche* (in inglese *elliptic curve primality testing*, abbreviato in ECPT) sono basati su alcune congetture della teoria analitica dei numeri, mentre il *test di Miller deterministico* è basato sulla *ipotesi di Riemann generalizzata*. Nel 2002 Agrawal, Kayal, e Saxena svilupparono un algoritmo di test deterministico efficiente, conosciuto come AKS dalle iniziali degli inventori [12].

Nonostante lo sviluppo del test AKS, questo è comunque inefficiente rispetto ai test probabilistici in quanto richiede uno sforzo computazionale estremamente maggiore. Questi ultimi hanno una certa probabilità p, detta *probabilità di errore*, di dichiarare come primo un numero composto. Tuttavia tale probabilità può essere ridotta a un numero arbitrariamente vicino a 0 ripetendo il test k volte: in tal caso la probabilità di errore diventa p^k. Scegliendo in modo opportuno il numero di ripetizioni k i test possono essere considerati ragionevolmente affidabili.

Per questi motivi si è scelto di utilizzare esclusivamente due test di primalità probabilistici nella generazione dei numeri primi. In particolare si sono usati il *test di Fermat* e il *test di Miller-Rabin*, che verranno presentati nel dettaglio nelle prossime sezioni.

3.3.1 Test di Fermat

Utilizzando il teorema di Fermat si può facilmente costruire un test di primalità probabilistico. La relazione 3.3 vale infatti per ogni primo p e per ogni intero $a \in \mathbb{Z}_q$, e dunque se si vuole testare la primalità di un intero q bisognerebbe verificare l'equivalenza per ogni $a \in \mathbb{Z}_q$. É tuttavia sufficiente verificare che questa valga solo per un sottoinsieme di $\mathbb{Z}_q$, scegliendone solo k elementi: se l'equivalenza è violata anche una sola volta allora q sarà sicuramente un numero composto, altrimenti può essere dichiarato probabilmente primo, o *pseudoprimo di Fermat*. Poiché si può mostrare che l'equivalenza vale banalmente per $a = 1, q-1$ per ogni intero q, si può restringere la ricerca a $2 \leq a \leq q-2$: tali valori vengono estratti casualmente.

Algoritmo 4. Test di Fermat

1: **for** $i = 1, ..., k$ **do**
2: Si sceglie un intero a casualmente tale che $2 \leq a \leq q - 2$
3: Si calcola $r \leftarrow a^{q-1} \mod q$
4: **if** $r \neq 1$ **then**
5: **return** `composto`
6: **end if**
7: **end for**
8: **return** `pseudoprimo`

Limitazioni

Se il test identifica un numero come composto, allora questo lo sarà certamente. Se invece identifica un numero come primo non si ha alcuna prova della correttezza di tale affermazione. Scegliendo opportunamente il valore di k si può ridurre la probabilità di errore, tuttavia questa non può essere portata arbitrariamente vicina a zero.

Esistono infatti alcuni interi composti, detti *numeri di Carmichael* q, tali che la relazione 3.3 è soddisfatta per ogni intero $a \in \mathbb{Z}_q^*$, ovvero per ogni a coprimo a q. Se allora q è un numero di Carmichael gli unici interi $a \in \mathbb{Z}_q$ che violano l'equivalenza sono quelli non coprimi a q. Se i fattori primi che compongono q sono tutti grandi, allora con grande probabilità il test di Fermat dichiarerà q primo anche con un numero di ripetizioni k grande.

Esempio:
Il numero di Carmichael più piccolo che si conosca è 561. Questo può essere fattorizzato in $561 = 3 \cdot 11 \cdot 17$, e si può vedere che $a^{560} \equiv 1 \pmod{561}$ per ogni $a \in \mathbb{Z}_{561}^*$. Gli unici interi modulo 561 non coprimi a 561 sono i multipli di 3, 11 e 17: questi sono solo 240 sui 560 possibili valori di a. Eseguendo un test di Fermat con 100 ripetizioni questo viene comunque dichiarato composto: il test di Fermat fallisce per numeri di Carmichael molto più grandi.

A causa dell'esistenza di tali numeri, il test di Fermat non è affidabile nel determinare la primalità di un intero q. I numeri pseudoprimi determinati da tale test sono detti per questo motivo *pseudoprimi deboli* e contengono al loro interno sia i numeri pseudoprimi che i numeri di Carmichael.

3.3.2 Test di Miller-Rabin

Partendo sempre dal teorema di Fermat è possibile sviluppare un nuovo test di primalità che non sia affetto dalla presenza dei numeri di Carmichael. Tale test è detto *test di Miller-Rabin* dal cognome degli sviluppatori [13][14].

Sia $q \geq 3$ un intero dispari, allora si può sempre fattorizzare la più grande potenza di 2 da $q-1$:

$$q - 1 = 2^s d \tag{3.7}$$

con $s \geq 1$ intero e d dispari. In tal modo utilizzando il binomio notevole $x^2 - 1 = (x-1)(x+1)$ si può fattorizzare ripetutamente $a^{q-1} - 1$ per ogni intero positivo a:

$$a^{q-1} - 1 = a^{2^s d} - 1 = (a^d - 1)(a^d + 1)(a^{2d} + 1)(a^{4d} + 1) \cdots (a^{2^{s-1} d} + 1) \tag{3.8}$$

Se q è un numero primo e $a \in [1, q-1]$ vale inoltre il teorema di Fermat, e dunque sostituendovi la fattorizzazione:

$$a^{q-1} - 1 = (a^d - 1)(a^d + 1)(a^{2d} + 1)(a^{4d} + 1) \cdots (a^{2^{s-1} d} + 1) \equiv 0 \pmod{q} \tag{3.9}$$

Usando le proprietà dell'operazione modulo, si mostra che l'equivalenza è verificata se è verificata almeno una delle equivalenze:

$$a^d \equiv \pm 1 \pmod{q} \tag{3.10}$$

$$a^{2^i d} \equiv -1 \pmod{q} \tag{3.11}$$

per qualche $i = 1, ..., s-1$.

Per un numero primo q è verificata almeno una delle equivalenze 3.10 per ogni intero $a \in \mathbb{Z}_q$. Se invece tutte le equivalenze sono violate anche per un solo a, q sarà sicuramente composto.

Come per il test di Fermat vengono scelti casualmente k valori di a nell'intervallo $[2, q-2]$, in quanto per $a = 1, q-1$ le equivalenze valgono banalmente per ogni q.

Nell'algoritmo si è fatto uso della proprietà dell'operazione modulo, per cui $a \equiv -1 \pmod{m}$ se e solo se $a \mod m = q - 1$.

Anche in questo caso se un intero viene identificato come composto allora lo sarà sicuramente, mentre se viene identificato come primo c'è una probabilità non nulla che questo sia in realtà composto.

Algoritmo 5. Test di Miller-Rabin

Si scrive $q - 1 = 2^s d$
for $i = 1, ..., k$ **do**
    Si sceglie un intero a casualmente tale che $2 \leq a \leq q - 2$
    Si calcola $x \leftarrow a^d \mod q$
    if $x = 1$ `or` $x = q - 1$ **then**
        Si ritorna a 3
    end if
    for $r = 1, ..., s - 1$ **do**
        Si calcola $x \leftarrow x^2 \mod q$
        if $x = q - 1$ **then**
            Si ritorna a 3
        end if
    end for
    return `composto`
end for
return `pseudoprimo`

Il test di Miller-Rabin è tuttavia più forte di quello di Fermat, in quanto non presenta il problema dei numeri di Carmichael: non si verifica più infatti la sola relazione 3.3, che nel caso di tali numeri vale per ogni $a \in \mathbb{Z}_q^*$, ma l'insieme di relazioni 3.10. I numeri pseudoprimi così generati vengono allora detti *pseudoprimi forti*.

3.3.3 Costruzione di un test di primalità efficiente

Per quanto mostrato nelle sezioni 3.3.1 e 3.3.2 i test di Fermat e Miller-Rabin sono in grado di dichiarare con certezza solo se un numero è composto: per tale motivo sono più propriamente detti *test di compostezza*.

Si è inoltre mostrato come il test di Fermat non sia affidabile in quanto restituisce dei numeri pseudoprimi deboli, di cui fanno parte anche i numeri di Carmichael. Al contrario il test di Miller-Rabin restituisce dei numeri pseudoprimi forti.
Si può tuttavia osservare che il test di Fermat richiede uno sforzo computazionale molto minore, in quanto richiede lo svolgimento di al massimo k elevamenti a potenza modulari, mentre il test di Miller-Rabin ne richiede al massimo $k \cdot s$.
Per tale motivo si è costruito un test di primalità efficiente combinando i due test:

1. In primo luogo il candidato q viene testato con un test di Fermat con numero di ripetizioni k_F. Se k_F è sufficientemente grande, se il test identifica q come pseudoprimo debole si può essere ragionevolmente sicuri che q sia o un numero

primo o un numero di Carmichael, altrimenti se q è identificato come composto allora lo sarà certamente.

2. Se il test di Fermat identifica q come un numero pseudoprimo debole, si esegue successivamente un test di Miller-Rabin con numero di ripetizioni k_M. Se k_M è sufficientemente grande, se il test identifica q come pseudoprimo forte allora si può essere ragionevolmente sicuri che q sia un numero primo, altrimenti se q è identificato come composto allora lo sarà certamente, e in aggiunta si può essere ragionevolmente sicuri che sia un numero di Carmichael.

Per rendere efficiente il test bisogna allora scegliere in modo opportuno k_F e k_M. Si può mostrare che la probabilità di errore del test di Fermat con una sola ripetizione è minore di $1/2$ (dove i numeri di Carmichael sono considerati come pseudoprimi deboli), mentre quella del test di Miller-Rabin è minore di $1/4$ [13]. Allora le probabilità di errore dopo k_F e k_M ripetizioni rispettivamente sono minori di $(1/2)^{k_F}$ e $(1/4)^{k_M}$. In teoria k_F e k_M potrebbero essere scelti grandi a piacere in modo da avere una probabilità di errore più bassa possibile. Tuttavia questo non è efficiente, perché richiederebbe uno sforzo computazionale troppo grande. Bisogna inoltre considerare la presenza di errori hardware, che occorrono in qualsiasi occasione in un computer. L'informazione del risultato del test è infatti un valore booleano che può essere rappresentato da un singolo bit (per esempio 1 se il numero testato è primo, 0 se è composto). Tale bit sarà mantenuto in una memoria volatile per un certo lasso di tempo. Se durante questo periodo un raggio cosmico interagisce con il sito fisico in cui il bit è memorizzato (per esempio un insieme di transistor) il valore del bit può essere cambiato: se per esempio il test identificava il numero testato come composto, questo sarà identificato come primo. Si può stimare la probabilità di tale processo nell'ordine di 10^{-24} [@15]. Se si scelgono allora $k_F = 100$ e $k_M = 50$ le probabilità di errore dei due test saranno inferiori a 10^{-31} e quindi si può essere ragionevolmente sicuri che un eventuale errore dei test sia causato da un errore hardware.

D'ora in poi si indicherà l'applicazione di tale test a un candidato q con:

$$T(q) = \begin{cases} \texttt{composto} \\ \texttt{pseudoprimo} \end{cases} \tag{3.12}$$

3.4 Algoritmi di generazione di numeri primi tradizionale

Utilizzando un generatore di numeri casuali e il test di primalità descritto in sezione 3.3.3 si possono costruire due diversi algoritmi per la generazione di numeri primi. Al fine di ottenere numeri primi utilizzabili nella generazione di chiavi RSA questi dovranno essere distribuiti uniformemente nell'intervallo $[q_{\text{min}}, q_{\text{max}}]$, dove generalmente $q_{\text{min}} = \sqrt{2^{2n-1}+1}$ e $q_{\text{max}} = 2^n - 1$ con $2n$ la lunghezza in bit della chiave.

3.4.1 Algoritmo di generazione casuale

L'algoritmo più semplice da costruire prevede l'estrazione di un numero intero casuale nell'intervallo voluto. L'intero così estratto viene testato con il test di primalità. Il procedimento viene ripetuto fino a quando non si trova un numero primo. L'algoritmo può essere reso più efficiente estraendo solo numeri dispari.

Algoritmo 6. Generatore casuale di numeri primi

1: **while** T(q)=`composto` **do**
2: Si estrae un intero dispari q casualmente tale che $q_{\text{min}} \leq q \leq q_{\text{max}}$
3: **end while**
4: **return** q

Limitazioni

Tale algoritmo è in grado di restituire numeri primi distribuiti uniformemente in tutto l'intervallo desiderato solo se il generatore di numeri casuali è uniforme nell'intervallo.

Inoltre la generazione di numeri casuali richiede uno sforzo non trascurabile (si veda appendice A.1). Tale metodo è dunque poco efficiente, perché richiede la generazione di un grande numero di numeri casuali e l'applicazione di un grande numero di test di primalità.

3.4.2 Algoritmo PRIMEINC

L'algoritmo precedente può essere reso più efficiente diminuendo drasticamente il numero di estrazione di numeri casuali, utilizzando una ricerca incrementale a

seguito della generazione di un'unico numero casuale. L'algoritmo fu analizzato per la prima volta da Brandt e Damgård [16] e venne denominato *PRIMEINC*.

Algoritmo 7. Generatore PRIMEINC

1: Si estrae un intero dispari q casualmente tale che $q_{\text{min}} \leq q \leq q_{\text{max}}$
2: **while** T(q)=`composto` **do**
3: Si calcola $q \leftarrow q + 2$
4: **if** $q > q_{\text{max}}$ **then**
5: Si pone q come il più piccolo numero dispari maggiore o uguale a q_{min}
6: **end if**
7: **end while**
8: **return** q

Limitazioni

Nonostante con tale algoritmo sia stato ridotto drasticamente il numero di estrazioni di numeri casuali, il numero di test di primalità eseguiti rimane comunque alto, in quanto i candidati q non sono scelti in maniera efficiente. I numeri primi hanno infatti una densità molto bassa, e dunque in media saranno necessarie diverse ripetizioni del ciclo **while** prima di estrarre un numero primo. A ogni ripetizione vengono eseguite al massimo $k_F + k_M s$ elevamenti a potenza modulare e $k_F + k_M$ estrazioni di numeri casuali per l'esecuzione dei test, richiedendo quindi uno sforzo computazionale non banale.

3.5 Generazione efficiente di numeri primi

Al fine di generare in modo efficiente numeri primi uniformemente distribuiti nell'intervallo $[q_{\text{min}}, q_{\text{max}}]$ è fondamentale generare dei candidati q che soddisfino alcune proprietà che li rendono primi con una probabilità maggiore, cosicché possa essere diminuito il numero di test di primalità eseguiti in media per generare un numero primo. Questo può essere fatto richiedendo che q sia coprimo a un numero intero composto m sufficientemente grande e che contenga un grande numero di fattori primi diversi. Il metodo che verrà illustrato di seguito è stato sviluppato da Joye e Paillier e riassunto negli articoli [17] e [18], per essere in seguito brevettato nel 2006 dall'azienda Gemplus [3].

L'algoritmo si avvale di tutte le proprietà dell'aritmetica modulare descritte nelle sezioni 2.3.1 e 3.1. Inoltre tale metodo presuppone la conoscenza dei primi numeri

primi: questi possono essere facilmente calcolati, per esempio usando il crivello di Eratostene descritto in sezione 3.2, oppure si possono trovare tabulati.

L'algoritmo può essere schematizzato nel modo seguente:

1. Si ricavano i parametri fondamentali l e m. I numeri primi generati dall'algoritmo saranno estratti nell'intervallo $[l, l+m-1] \subseteq [q_{\text{min}}, q_{\text{max}}]$, dunque i due parametri sono numeri interi scelti in modo tale che l'intervallo di generazione approssimi nel migliore modo possibile l'intervallo desiderato. In particolare devono valere:
$$l \gtrsim q_{\text{min}} \qquad m \lesssim q_{\text{max}} - q_{\text{min}} + 1 \tag{3.13}$$
Inoltre i candidati verranno estratti nel gruppo dei numeri coprimi a m $\mathbb{Z}_m^*$. Per rendere l'algoritmo il più efficiente possibile, in modo da diminuire il numero medio di test di primalità eseguiti per generare un numero primo, bisogna richiedere che m contenga un grande numero di fattori primi diversi. Questo è equivalente a richiedere che il rapporto $\varphi(m)/m$ sia il più piccolo possibile, in quanto quest'ultimo è il rapporto fra il numero di elementi coprimi a m nell'anello degli interi modulo m $\mathbb{Z}_m$ e il numero di elementi nell'anello stesso. Se infatti m è composto dal più grande numero possibile di fattori primi diversi, tutti i multipli di tali fattori primi non faranno parte di $\mathbb{Z}_m^*$ e quindi il rapporto sarà minimo.

2. Si estrae casualmente un intero x dal gruppo $\mathbb{Z}_m^*$. Questo viene fatto a partire dall'estrazione di un numero casuale in $\mathbb{Z}_m$ tramite un generatore di numeri casuali. In questo caso si è usato il generatore di stringhe di bit casuali RandomPower [@19][20].

3. Si costruisce il candidato $q = l + x$. Poiché $l \gtrsim q_{\text{min}}$ e $x \in \mathbb{Z}_m^*$ con $m \lesssim q_{\text{max}} - q_{\text{min}} + 1$, tale candidato si trova nell'intervallo desiderato e soddisfa inoltre le proprietà di coprimalità richieste: sarà allora un numero primo con probabilità maggiore rispetto ai candidati generati dagli algoritmi tradizionali.

4. Il candidato viene testato tramite il test di sezione 3.3.3 $T(q)$. Se questo viene identificato come numero primo il processo si conclude, altrimenti si procede generando un altro elemento $x \in \mathbb{Z}_m^*$. Tuttavia poiché il processo di generazione di un'unità è dispendioso dal punto di vista computazionale, si può eseguire una sorta di ricerca incrementale sulla falsa riga dell'algoritmo *PRIMEINC*. Sfruttando le proprietà del gruppo moltiplicativo $\mathbb{Z}_m^*$ si può scegliere opportunamente un suo elemento $a \in \mathbb{Z}_m^*$ e generare così una nuova unità come $x' = ax \mod m$. $\mathbb{Z}_m^*$ è infatti chiuso sotto moltiplicazione modulare,

essendo un gruppo moltiplicativo. L'unità a deve essere scelta in modo da evitare di ritrovare sempre gli stessi elementi $x \in \mathbb{Z}_m^*$. Questo può essere fatto per esempio scegliendo la più piccola unità in $\mathbb{Z}_m^*$. Dopo aver generato una nuova unità x, si costruisce un nuovo candidato $q' = x' + l$, e questo viene nuovamente testato tramite il test di primalità $T(q)$. Il processo è ripetuto fino a quando non viene generato un numero primo.

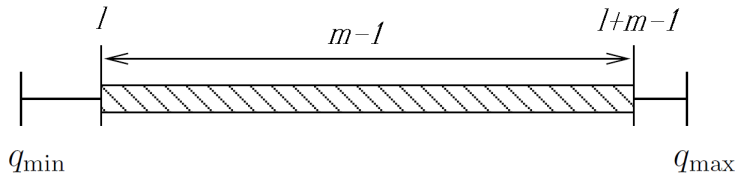


Fig. 3.1.: Schema dell'intervallo di generazione (rettangolo tratteggiato) $[l, l+m-1]$ in riferimento all'intervallo desiderato (linea continua) $[q_{\min}, q_t extmax]$.

La parte 1, che d'ora in poi sarà denominata *fase di configurazione*, deve essere svolta una volta sola e quindi può essere effettuata su un computer. Le parti 2, 3 e 4 fanno invece parte della *fase di generazione* e devono essere implementate sulla chiave elettronica che servirà a generare la chiave RSA.

3.5.1 Fase di configurazione

La fase di configurazione è fondamentale al fine di ottenere un algoritmo di generazione di numeri primi efficiente. L'efficienza è infatti determinata dalla scelta dei parametri l e m.

In tale scelta si utilizza un parametro di qualità ε tale che $0 < \varepsilon \leq 1$. Come si vedrà questo permette di determinare quanto bene l'intervallo di generazione $[l, l+m-1]$ approssima l'intervallo desiderato $[q_{\min}, q_{\max}]$: più piccolo è ε migliore è tale approssimazione. Un valore tipico di ε è 10^{-3}, anche se nell'implementazione si è scelto di usare $\varepsilon = 10^{-5}$ per una migliore approssimazione.

Per generare i parametri l e m bisogna scegliere un intero $\Pi = \prod_{i=1}^{k} p_i$ come il prodotto dei primi k numeri primi. Tale intero deve essere scelto in modo che esistano due interi v e w tali che siano soddisfatte le seguenti proprietà:

$$1 - \varepsilon < \frac{w\Pi - 1}{q_{\max} - q_{\min}} \leq 1 \tag{P1}$$

$$v\Pi \geq q_{\min} \tag{P2}$$

$$(v + w)\Pi - 1 \leq q_{\text{max}} \tag{P3}$$

$$\varphi(\Pi)/\Pi \qquad \text{é più piccolo possibile} \tag{P4}$$

Definendo allora $l = v\Pi$ e $m = w\Pi$ le proprietà P1, P2 e P3 assicurano che l'intervallo di generazione approssimi quello desiderato nel miglior modo possibile. In particolare P1 assicura che la lunghezza dell'intervallo di generazione $m - 1$ sia più vicina possibile alla lunghezza dell'intervallo desiderato $q_{\text{max}} - q_{\text{min}}$: un valore di ε più piccolo rende le due più vicine. La proprietà P4 assicura invece che m contenga il maggior numero possibile di fattori primi diversi, rendendo così l'algoritmo efficiente.

Nella pratica tali parametri sono scelti generando tutti i possibili interi $\Pi < q_{\text{max}} - q_{\text{min}}$. Dopodiché viene scelto quello con il valore di $\varphi(\Pi)/\Pi$ minore e si controlla se esistono due interi v e w che soddisfano le proprietà P1, P2 e P3. Se tali interi non esistono per questo valore di Π, si sceglie quello con il secondo valore di $\varphi(\Pi)/\Pi$ più basso. Il processo è ripetuto fino a quando non si riescono a scegliere dei valori per v e w.

3.5.2 Fase di generazione

Nella fase di generazione risulta cruciale la generazione dell'unità $x \in \mathbb{Z}_m^*$ a partire da un numero casuale. A tal scopo il teorema di Carmichael permette di determinare in maniera semplice quando un elemento di $\mathbb{Z}_m$ è un'unità. L'algoritmo di generazione consiste dunque nel generare casualmente un numero intero $x \in \mathbb{Z}_m$ modulo m. Se questo verifica l'identità 3.4 allora è un'unità, altrimenti si procede con una ricerca incrementale, passando all'intero successivo $x + 1$. Il processo viene ripetuto fino a quando non viene generato $x \in \mathbb{Z}_m^*$.

Algoritmo 8. Generatore di unità

1: Si calcola la funzione di Carmichael $\lambda(m)$
2: Si genera un numero casuale x tale che $1 \leq x \leq m - 1$
3: **while** $x^{\lambda(m)} \mod m \neq 1$ **do**
4: Si calcola $x \leftarrow x + 1$
5: **end while**
6: **return** x

La funzione di Carmichael è calcolabile facilmente, in quanto $m = w\Pi$: la fattorizzazione di Π è conosciuta per costruzione, mentre w è un intero piccolo e quindi

facilmente fattorizzabile con funzioni di libreria implementate nella maggior parte dei linguaggi di programmazione. Conosciuta la fattorizzazione di w e Π quella di m è ricavabile banalmente, e dunque $\lambda(m)$ si calcola utilizzando la relazione 3.5.

Infine per generare efficientemente un numero primo, bisogna scegliere in modo opportuno una unità $a \in \mathbb{Z}_m^*$. A tale scopo si è scelto di prendere l'elemento più piccolo del gruppo moltiplicativo, utilizzando ancora il teorema di Carmichael per identificarlo.

Algoritmo 9. Generatore di numeri primi efficiente

1: Si genera un'unità $x \in \mathbb{Z}_m^*$
2: Si calcola $q \leftarrow x + l$
3: **while** $T(q) =$ `composto` **do**
4: Si calcola $x \leftarrow ax \mod m$
5: Si calcola $q \leftarrow x + l$
6: **end while**
7: **return** q

L'algoritmo può essere migliorato ulteriormente imponendo $a = 2$. In questo modo infatti l'operazione $x \leftarrow 2x \mod m$ è banale: la moltiplicazione $2x$ corrisponde all'aggiunta di un bit 0 a x, ovvero è equivalente all'operazione $x << 1$, mentre essendo sicuramente $2x$ minore di $2m$ l'operazione modulo corrisponde eventualmente alla sottrazione $2x - m$ nel caso in cui $2x \geq m$.
Tuttavia per come si è costruito Π, e dunque m, $a = 2$ non è un'unità in quanto è un fattore primo di m. Questo può essere facilmente corretto imponendo nella fase di configurazione che Π sia il prodotto dei primi k numeri primi escluso 2, e che w e v siano interi dispari. A tal scopo la proprietà P2 deve essere cambiata in:

$$v\Pi + 1 \geq q_{\text{min}} \tag{P2'}$$

m e l rimangono invece definiti come in precedenza, e dunque saranno numeri dispari non avendo più il fattore primo 2.
Poichè m non ha più come fattore primo 2, i candidati q potranno essere questa volta anche numeri pari. Per evitare di svolgere test inutili, viene prima controllata la parità di ogni candidato $q = x + l$: se questo risulta essere pari, si costruire facilmente un nuovo candidato dispari come $q = m - w + l$. Infatti usando le proprietà dell'operazione modulo $m - x + l \equiv m + (x + l) \pmod 2$, in quanto nell'anello degli interi modulo 2 $-x \equiv x \pmod 2$. Allora usando le proprietà dell'addizione modulare, essendo $x + l$ pari e m dispari, si ha $m - x + l \equiv m \equiv 1$, ovvero $m - x + l$ è dispari.

Algoritmo 10. Generatore di numeri primi efficiente migliorata

1: Si genera un'unità $x \in \mathbb{Z}_m^*$
2: Si calcola $q \leftarrow x + l$
3: **while** $T(q) =$ `composto` **do**
4: Si calcola $x \leftarrow ax \mod m$
5: Si calcola $q \leftarrow x + l$
6: **if** q è pari **then**
7: Si calcola $q \leftarrow m - x + l$
8: **end if**
9: **end while**
10: **return** q

3.5.3 Estensibilità dell'intervallo

Il metodo di configurazione presentato permette di calcolare facilmente i parametri m, l e $\lambda(m)$. Si può mostrare che nel caso della generazione di numeri primi per la crittografia RSA si possono facilmente ricavare i parametri per chiavi di diverse lunghezze a partire dai parametri per chiavi di una lunghezza fissata. In particolare, si è mostrato che al fine di generare chiavi RSA a $2n$-bit bisogna estrarre numeri primi nell'intervallo $[q_{\text{min}}, q_{\text{max}}]$ dove $q_{\text{min}} = \sqrt{2^{2n-1}+1}$ e $q_{\text{max}} = 2^n - 1$. Gli estremi possono essere approssimati con $q_{\text{min}} \approx 2^{n-1/2}$ e $q_{\text{max}} = 2^n$. Si vuole mostrare che i parametri $m(n)$, $l(m)$ e $\lambda(m(n))$ per la generazione di chiavi RSA a $2n$-bit sono facilmente ricavabili dai parametri $m(n_0)$, $l(n_0)$ e $\lambda(m(n_0))$ per la generazione di chiavi RSA a $2n_0$-bit per ogni $n > n_0$.

Usando le approssimazioni per q_{min} e q_{max} si può facilmente vedere che $q_{\text{min}}(n) \approx q_{\text{min}}(n_0)2^{n-n_0}$ e $q_{\text{max}}(n) \approx q_{\text{max}}(n_0)2^{n-n_0}$. Si considerino allora le trasformazioni:

$$\begin{cases} \Pi(n) = \Pi(n_0) \\ v(n) = v(n_0)2^{n-n_0} \\ w(n) = w(n_0)2^{n-n_0} \end{cases} \tag{3.14}$$

dove $m(n_0) = w(n_0)\Pi(n_0)$, $l(n_0) = v(n_0)\Pi(n_0)$, $m(n) = w(n)\Pi(n)$ e $l(n) = l(n)\Pi(n)$. Si può mostrare che tali trasformazioni preservano le propreità P1, P2 e P3 con $\varepsilon(n) = \varepsilon(n_0)$.

Segue allora che per ogni $n > n_0$ si possono ricavare i parametri come $m(n) = m(n_0)2^{n-n_0}$ e $l(n) = l(n_0)2^{n-n_0}$. La funzione di Carmichael è anch'essa calcolabile facilmente, e vale $\lambda(m(n)) = \lambda(m(n_0))2^{n-n_0}$. I parametri così trovati non soddisfano sempre la condizione P4 e quindi l'algoritmo perde in efficienza. Tuttavia questo è compromesso sicuramente accettabile se comparato con il guadagno di memoria

ottenuta: è infatti possibile salvare su una piccola memoria incorporata nella scheda elettronica per la generazione di chiavi RSA i parametri per una sola lunghezza n_0 e generare numeri primi per qualsiasi lunghezza $n > n_0$.

3.5.4 Implementazione dell'algoritmo

Si è scelto di implementare l'algoritmo su un personal computer tramite il linguaggio di programmazione *Python*. Questo infatti permette di gestire numeri interi di lunghezza arbitraria, a differenza di altri linguaggi come *Matlab* che permettono di gestire numeri interi fino a massimo 64-bit.

Tutti gli algoritmi presentati in questa tesi sono stati implementati efficacemente. In particolare, al fine di ottenere una valutazione dell'efficienza dell'algoritmo presentato in sezione 3.5 è stato implementato anche l'algoritmo *PRIMEINC* ed è stato eseguito un confronto fra i due.

Per generare le unità $x \in \mathbb{Z}_m^*$ si sono usate delle stringhe di bit causali estratte dal generatore RandomPower, mentre per i test di primalità si è usata la funzione *random.randrange* della libreria *random* di Python [@21] per generare numeri interi pseudocasuali. Tale scelta è stata presa a causa della limitata memoria in dotazione al computer utilizzato ($6\,GB$ di RAM) che rendeva il processo di caricamento in memoria delle stringhe casuali troppo lento se effettuato troppe volte. In questo modo da ogni stringa di bit casuali a n-bit si è potuto estrarre un numero primo a n-bit.

3.6 Confronto degli algoritmi di generazione e valutazione dell'efficienza

In questa sezione verrà confrontato l'algoritmo di generazione efficiente migliorato con l'algoritmo *PRIMEINC*. In questo modo si potrà valutare l'efficienza del primo.

In primo luogo si può valutare il tempo medio richiesto per generare un numero primo dai due algoritmi. In figura 3.2 sono mostrati i tempi medi di generazione di un numero primo per i due algoritmi, al variare della lunghezza in bit dei numeri primi generati . Come si può vedere il guadagno nell'utilizzo dell'algoritmo efficiente è solo di un fattore 2. Inoltre è mostrato anche il tempo medio richiesto per generare

un'unità nell'algoritmo efficiente: come si può osservare, questo è quasi ininfluente sul tempo totale di generazione del numero primo.

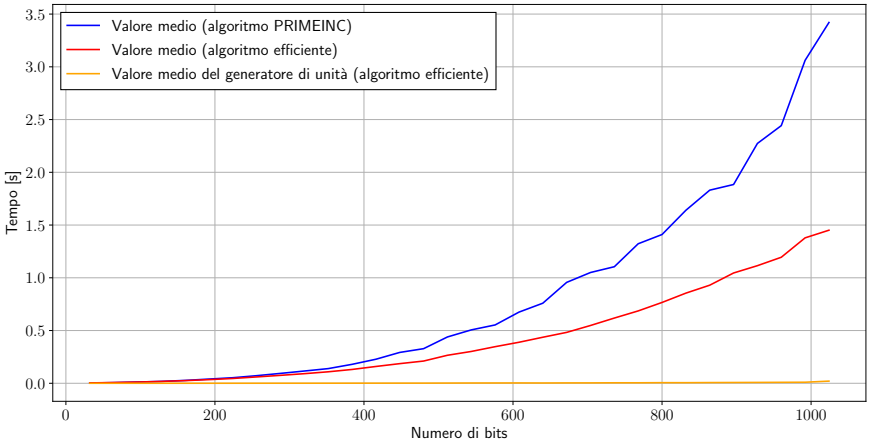


Fig. 3.2.: Tempo medio di generazione di un numero primo per l'algoritmo efficiente migliorato (in rosso) e per l'algoritmo *PRIMEINC* (in blu). Per l'algoritmo efficiente è anche mostrato in verde il tempo medio richiesto per generare un'unità. Le medie sono calcolate su 500 valori diversi.

Il tempo medio di generazione non può allora essere utilizzato come indicatore dell'efficienza dell'algoritmo. Questo dipende infatti da molti fattori esterni, come la velocità del processore nell'esecuzione di particolari calcoli. In effetti la maggior parte dei criptoprocessori installati nelle schede elettroniche per la generazione di chiavi RSA sono costruiti in modo da eseguire molto velocemente operazioni come l'elevamento a potenza modulare (che nel nostro caso è l'operazione che richiede più tempo computazionale), ma al contempo richiedono un tempo molto più elevato di un normale processore per eseguire operazioni come la somma o la sottrazione.

Prima di studiare un nuovo parametro per la classificazione dell'efficienza dell'algoritmo, si può osservare più in dettaglio la distribuzione dei tempi richiesti per la generazione dei numeri primi. In figura 3.3 sono mostrate le distribuzioni dei tempi di generazione per 500 numeri primi a 1024-bit. Come si può vedere queste sono molto simili per i due algoritmi, con un picco molto a sinistra e una coda a destra lunga.

Poiché i tempi di generazione non sono dei buoni indicatori per l'efficienza dell'algoritmo, si deve studiare una nuova metrica che permetta di definire tale efficienza e che sia indipendente dai fattori esterni, quali il particolare tipo di processore utilizzato. A tal proposito si può notare che il tempo di generazione dipenderà dal numero di processi eseguiti dall'algoritmo prima di generare un numero primo.

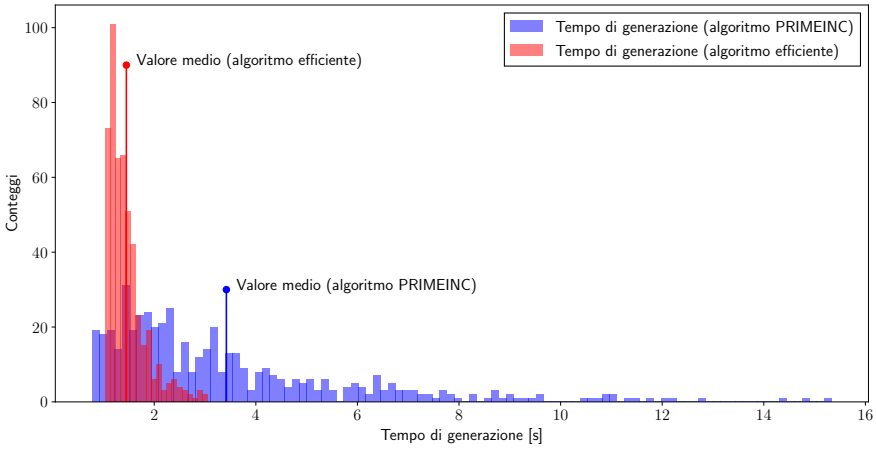


Fig. 3.3.: Distribuzione dei tempi di generazione di un numero primo a 1024-bit per l'algoritmo efficiente migliorato (in rosso) e per l'algoritmo *PRIMEINC* (in blu). La linea verticale rossa indica il tempo medio di generazione dell'algoritmo efficiente, mentre quella blu il valore medio dell'algoritmo *PRIMEINC*.

In particolare il processo più dispendioso è l'elevamento a potenza modulare, che compare più volte in entrambi gli algoritmi:

- Per entrambi gli algoritmi, l'esecuzione di un singolo test di primalità richiede al massimo $k_F + k_M s$ elevamenti a potenza modulare, dove k_F e k_M sono il numero di ripetizioni nel test di Fermat e Miller-Rabin rispettivamente (in questo caso 100 e 50 rispettivamente), e s è l'esponente della potenza di 2 nella fattorizzazione di $q - 1$, come descritto in sezione 3.3.2.
- Per il solo caso dell'algoritmo efficiente la generazione di un'unità richiede tanti elevamenti a potenza modulare quante sono le ripetizioni del ciclo **while** nell'algoritmo. Tale algoritmo viene eseguito una sola volta per ogni numero primo generato.

Siano allora N_T il numero medio di test di primalità eseguiti, ovvero il numero di candidati testati, e N_U il numero medio di ripetizioni nel generatore di unità. Il generatore *PRIMEINC* richiederà in media l'esecuzione di $N_P = N_T(k_F + k_M \bar{s})$ elevamenti a potenza modulare, mentre il generatore efficiente richiederà in media l'esecuzione di $N_E = N_T(k_F + k_M \bar{s}) + N_U$ elevamenti a potenza modulare. Il valore $\bar{s}$ è la media empirica dell'esponente s. In figura 3.4 è rappresentato il valore di N_T al variare della lunghezza in bit dei numeri primi generati per entrambi gli algoritmi, mentre in figura 3.5 è rappresentato il valore di N_U per l'algoritmo efficiente, al variare della lunghezza in bit dei numeri primi generati.

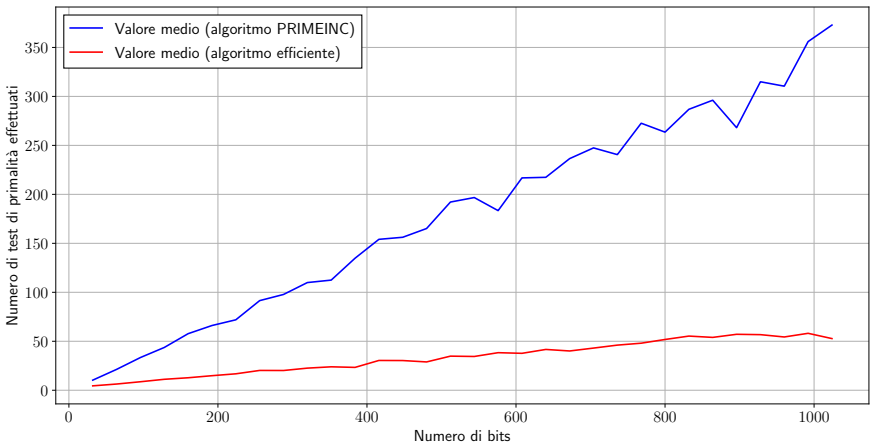


Fig. 3.4.: Numero medio di test di primalità eseguiti per generare un numero primo per l'algoritmo efficiente migliorato (in rosso) e per l'algoritmo *PRIMEINC* (in blu).

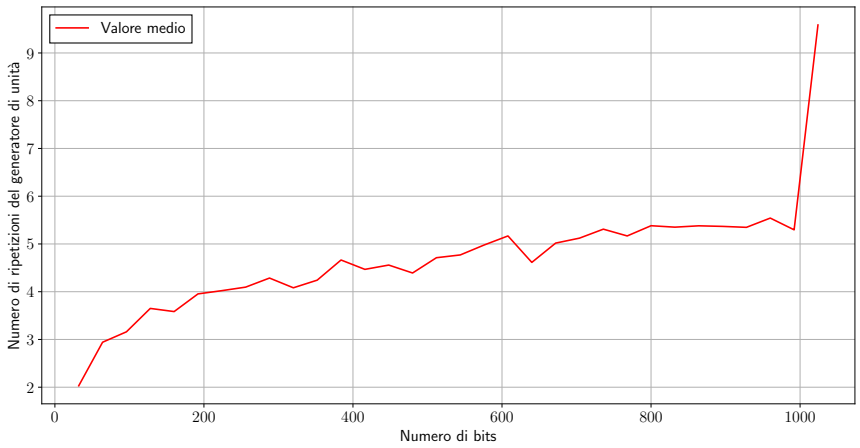


Fig. 3.5.: Numero medio di ripetizioni nell'algoritmo di generazione di un'unità per l'algoritmo efficiente.

Come si può vedere il valore di N_T per l'algoritmo efficiente è circa 5 volte minore di quello per l'algoritmo *PRIMEINC*. Nel primo infatti si restringe la ricerca dei candidati ai soli numeri coprimi a m, diminuendo così i possibili valori da testare. Inoltre poiché il valore di N_U è estremamente piccolo se comparato a quello di N_T per l'algoritmo *PRIMEINC*, si comprende come l'algoritmo efficiente porti dei vantaggi non indifferenti nella generazione dei numeri primi. Utilizzando come indicatori dell'efficienza degli algoritmi i due valori N_E e N_P, essendo N_U trascurabile, si può osservare come $N_E \approx N_P/5$.

3.7 Conclusioni

Si è mostrato come la generazione di numeri primi richieda un grande sforzo computazionale. In particolare si sono mostrati due diversi algoritmi di generazione di numeri primi tradizionali, e si è presentato un algoritmo efficiente, basato su conoscenze profonde dell'aritmetica modulare. Tale algoritmo permette di diminuire il numero di test di primalità effettuati cercando i candidati fra i numeri coprimi a un intero m composto da un grande numero di diversi fattori primi.

Si è infine implementato l'algoritmo efficiente e si è confrontata la sua efficienza con un algoritmo tradizionale, mostrando come effettivamente il primo porti a dei vantaggi notevoli.

4 Conclusioni

Come visto nel capitolo 2 uno degli schemi fondamentali della crittografia, ovvero la crittografia RSA, richiede la generazione di grandi numeri primi affinché sia sicuro. Sorprendentemente, nonostante i numerosi studi matematici sui numeri primi, tutt'ora esistono pochi algoritmi di generazione, e quei pochi che esistono hanno delle prestazioni molto povere. Tutti questi algoritmi sono infatti basati sull'estrazione casuale di un intero, la cui primalità viene in seguito verificata tramite dei test di primalità.

Si è allora presentato un metodo che permette di migliorare efficacemente le prestazioni degli algoritmi di generazione, cercando i candidati solo fra i numeri coprimi a un intero m composto da un grande numero di fattori primi diversi. Questo riduce estremamente il numero di candidati da testare, rendendo quindi più efficiente l'algoritmo.

In particolare si è presentato un modo per unire i test di Fermat e di Miller-Rabin, per costruire un test di primalità $T(q)$ più efficiente. Questo infatti esegue prima un test di Fermat con 100 ripetizioni, che richiede uno sforzo computazionale minore ma restituisce solo numeri pseudoprimi deboli. In questo modo i candidati composti vengono velocemente esclusi. I candidati che passano il test di Fermat invece vengono in seguito testati con un test di Miller-Rabin con 50 ripetizioni, in modo da identificare i numeri primi con un livello di confidenza accettabile. La probabilità che un numero composto passi il test $T(q)$ vendendo identificato come numero primo è inferiore a 10^{-31}, molto minore della probabilità di errore hardware, stimata nell'ordine di 10^{-24}.

L'algoritmo di generazione efficiente è stato infine implementato su un personal computer tramite il linguaggio di programmazione Python, ed è stato confrontato con l'algoritmo tradizionale *PRIMEINC*. Nonostante il tempo di generazione medio migliori solo di un fattore 2, si è mostrato come questo non possa essere usato come indicatore dell'efficienza dell'algoritmo, in quanto dipende fortemente da fattori esterni quali il tipo di processore utilizzato. Si è allora costruita una metrica considerando il numero di elevamenti a potenza modulari eseguiti in media dai due algoritmi, in quanto questa è l'operazione che richiede uno sforzo computazionale maggiore in presenza di numeri interi grandi. Come previsto, l'algoritmo efficiente

richiede in media l'esecuzione di un quinto degli elevamenti a potenza modulari richiesti dall'algoritmo *PRIMEINC*.

4.1 Possibili miglioramenti

Gli algoritmi di generazione di numeri primi e generazioni di chiavi RSA dovrebbero essere implementati direttamente in una scheda elettronica dotata di un crittoprocessore. Questi tipi di processori sono studiati apposta per eseguire in modo estremamente veloce certi calcoli come l'elevamento a potenza modulare, rendendo probabilmente ancora più efficiente l'algoritmo presentato. Inoltre devono soddisfare particolari richieste di sicurezza che impediscano a qualsiasi intruso di poter ricavare informazioni sensibili. I normali processori di un personal computer sono inadatti a tale scopo, perché soggetti a bug che possono essere sfruttati da un avversario.

Un futuro miglioramento del lavoro svolto, necessario per sfruttare al massimo l'algoritmo sviluppato, è dunque l'implementazione di quest'ultimo su una chiave elettronica. A tale scopo sarebbe interessante montare su tale scheda il generatore di numeri casuali RandomPower, in modo da poter utilizzare le stringhe di bit casuali generate in maniera efficiente. Utilizzando un opportuno processore, si potrebbero inoltre sfruttare le proprietà del calcolo in parallelo per aumentare ulteriormente le prestazioni. I processi che potrebbero essere parallelizzati sono per esempio i test di Fermat e Miller-Rabin, assieme alla generazione dell'unità.

Infine, allo scopo di costruire un prodotto spendibile nel mercato della cybersicurezza, si dovrebbe implementare la generazione della chiavi RSA direttamente nella scheda elettronica. L'articolo [18] suggerisce un metodo per sfruttare efficacemente l'algoritmo di generazione dei numeri primi a tale scopo.

Bibliografia

[1] R. Rivest; A. Shamir; L. Adleman. «A method for obtaining digital signatures and public-key cryptosystems». In: *Communications of the ACM* 26.1 (1983), pp. 96–99 (cit. alle pp. 1, 5).

[2] W. Stallings. *Cryptography and Network Security: Principles and Practice*. Pearson Education Limited, 2017 (cit. alle pp. 1, 3, 7, 8, 11, 14).

[3] M. Joye; P. Paillier. «Method for generating a random prime number within a predetermined interval». Brev. US 7,149,763 B2. Gemplus. 12 Dic. 2017 (cit. alle pp. 1, 22).

[4] A. Menezes; P. Van Oorschot; S. Vanstone. *Handbook of applied cryptography*. CRC Press, 1997 (cit. alle pp. 3, 15).

[5] W. Diffie; M. Hellman. «Multiuser cryptographic techniques». In: *AFIPS '76: Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*. 1976 (cit. a p. 5).

[8] P. W. Shor. «Algorithms for quantum computation: discrete logarithms and factoring». In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 1994, pp. 124–134 (cit. a p. 12).

[9] R. Crandall; C. Pomerance. *Prime Numbers: A Computational Perspective*. Springer-Verlag New York, 2001 (cit. a p. 13).

[10] P. Ribenboim. *The New Book of Prime Number Records*. Springer-Verlag New York, 1996 (cit. alle pp. 13, 14).

[11] R. D.Carmichael. *The Theory of Numbers*. John Wiley & Sons, 1914 (cit. a p. 14).

[12] M. Agrawal; N. Kayal; N. Saxena. «PRIMES is in P». In: *Annals of Mathematics* 160.2 (2004), pp. 781–793 (cit. a p. 16).

[13] G. Miller. «Riemann's hypothesis and tests for primality». In: *Journal of computer and system sciences* 13.3 (1976), pp. 300–317 (cit. alle pp. 18, 20).

[14] M. Rabin. «Probabilistic Algorithms for Primality Testing». In: *Journal of Number Theory* 12.1 (1980), pp. 128–138 (cit. a p. 18).

[16] J. Brandt; I. Damgård. «Probabilistic Algorithms for Primality Testing». In: *Brickell E.F. (eds) Advances in Cryptology — CRYPTO' 92. CRYPTO 1992. Lecture Notes in Computer Science* 740 (1993), pp. 358–370 (cit. a p. 22).

[17] M. Joye; P. Paillier; S. Vaudenay. «Efficient generation of prime numbers». In: *Koç Ç.K., Paar C. (eds) Cryptographic Hardware and Embedded Systems — CHES 2000. CHES 2000. Lecture Notes in Computer Science* 1965 (2000), pp. 340–345 (cit. a p. 22).

[18] M. Joye; P. Paillier. «Fast Generation of Prime Numbers on Portable Devices: An Update». In: *Goubin L., Matsui M. (eds) Cryptographic Hardware and Embedded Systems - CHES 2006. CHES 2006. Lecture Notes in Computer Science* 4249 (2006), pp. 160–173 (cit. alle pp. 22, 34).

[20] M. Caccia;et al. «In-silico generation of random bit streams». In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 980 (2020) (cit. a p. 23).

Webpages

[@6] RSA Security. *Factorization of RSA-250*. 1999. URL: https://lists.gforge.inria.fr/pipermail/cado-nfs-discuss/2020-February/001166.html (visitato il 30 gen. 2002) (cit. a p. 11).

[@7] RSA Security. *RSA Honor Roll*. 2020. URL: http://www.ontko.com/pub/rayo/primes/hr_rsa.txt (visitato il 28 feb. 2020) (cit. a p. 11).

[@15] Stack Overflow. *RSA and prime-generator algorithms*. 2010. URL: https://stackoverflow.com/questions/4159333/rsa-and-prime-generator-algorithms/4160517#4160517 (visitato il 11 nov. 2010) (cit. a p. 20).

[@19] Random Power. *Random Power Project*. 2020. URL: https://www.randompower.eu/ (cit. a p. 23).

[@21] Python. *random — Generate pseudo-random numbers*. URL: https://docs.python.org/3/library/random.html (cit. a p. 28).

Appendice

A

A.1 Generazione di numeri casuali

La sicurezza di molti sistemi crittografici, inclusa la crittografia RSA, è basata sulla capacità di generare numeri casuali. Questo può essere fatto a partire dalla generazione di una stringa di bit casuali, ovvero indipendenti uno dall'altro e ognuno con probabilità $1/2$ di essere 1 o 0. Infatti un intero uniformemente generato nell'intervallo $[1, m]$ può sempre essere ottenuto da una stringa di bit casuali di lunghezza $\log_2 n + 1$ convertendola in un intero. La generazione di bit casuali può essere ottenuta in due modi diversi:

Generazione di bit casuali Utilizzando una qualche sorgente fisica che abbia un comportamento casuale, si generano dei bit casuali. I numeri interi così generati vengono detti numeri casuali, e il metodo viene detto in inglese *True Random Number Generation* o *TRNG*.

Generazione di bit pseudocasuali Utilizzando un algoritmo deterministico si generano dei bit che, nonostante non sono casuali in quanto generati deterministicamente, sembrano tali. Il metodo viene detto in inglese *Pseudo Random Number Generation* o *PRNG*.

Negli algoritmi presentati vengono usati entrambi i metodi. In particolare si usa la generazione di numeri casuali nella generazione dell'unità. Questo viene fatto con il sistema RandomPower, il quale è in grado di generare un flusso di bit casuali analizzando le serie temporali di impulsi generati in un fotomoltiplicatore al silicio (SiPM dall'inglese *Silicon PhotoMultiplier*) posto al buio. Tali impulsi sono completamenti casuali, e dettati da fenomeni quantistici. Tutte le stringhe di bit utilizzate sono state testate seguendo le procedure necessarie a verificarne la casualità.