Università degli Studi di Padova

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Tesi di *Laurea Triennale* in Ingegneria Informatica

Analisi prestazionale dell'algoritmo RSA per sistemi embedded

Relatore
prof. Nicola Zingirian

Candidato
Leonardo Gemin

Anno Accademico 2019/2020

21 settembre 2020

Ai miei genitori,
pilastri della mia famiglia,
senza i quali non sarei qui, ora.

Ai miei nonni,
i miei più grandi *fan*,
che da sempre mi sostengono e mi voglio bene.

Al mio nonnino,
che dal silenzio più luminoso
sarebbe, ed è, orgoglioso di questo traguardo raggiunto.

A mio fratello,
a cui auguro tutto il bene di questo mondo,
a me stesso
e ai tanti nostri obiettivi da raggiungere.

"Per apsera ad astra"

Abstract

Con l'evolversi della tecnologia, oggetti sempre più piccoli necessitano soluzioni in grado di comunicare dati sensibili senza il timore che tali dati possano essere letti da persone non autorizzate.

Proprio per questo motivo, si è resa necessaria una reimplementazione del crittosistema RSA, letto in chiave *embedded*, ovvero in condizioni di scarsa memoria disponibile e in oggetti che hanno esclusivamente una o poche funzionalità.

Il crittosistema, sviluppato in linguaggio C, non fa uso di librerie esterne oltre a quelle standard. Questo è importante perché permette di avere l'esatta computazione del codice, in modo da poterne frammentare l'esecuzione nel caso in cui venisse eseguito su un dispositivo embedded.

Le varie funzioni sono state progettate per ottenere un giusto compromesso tra la complessità computazionale e l'utilizzo di memoria, per cui alcune procedure, pur presentando complessità non ottimali, si rivelano estremamente adatte al problema esposto. È il caso, ad esempio, della moltiplicazione, presentata nei capitoli sucessivi, caratterizzata da una complessità quadratica, molto più degli algoritmi più recenti, a partire da quello di Karatsuba, passando per la procedura di Toom-Cook e per l'algoritmo di Schönhage-Strassen, fino alla trasformata di Fürer.

In relazione alla complessità computazionale è il tempo. Dopo numerosi tentativi, manipolazioni e ristrutturazioni del codice, si è riusciti a far svolgere al calcolatore le operazioni per la generazione delle chiavi cifrate in tempi pressoché ragionevoli, come illustrato nei capitoli a seguire.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco degli algoritmi

1

L'algoritmo RSA

Pubblicato per la prima volta nel 1977, l'algoritmo RSA prende il nome dai suoi inventori, Ronald Rivest, Adi Shamir e Leonard Adleman. La sigla indica un sistema di *crittografia asimmetrica*, basato sull'esistenza di due chiavi distinte: una pubblica, utilizzata per effettuare la cifratura del messaggio; l'altra privata, usata per la decifratura dello stesso. La peculiarità che contraddistingue questo metodo di cifratura, e che sta alla base di tale algoritmo, è che, pur essendo a conoscenza di una delle due chiavi, non è possibile risalire a quella mancante. In questo modo viene garantita l'integrità del sistema.

Per realizzare un tale sistema crittografico, è necessario che ogni utente generi entrambe le chiavi e che renda pubblica la *chiave diretta*, mantenendo strettamente privata la *chiave inversa*.

L'affidabilità e la sicurezza dell'algoritmo RSA deriva dall'enorme mole di calcoli e dall'enorme dispendio, in termini di tempo, necessario per trovare la soluzione alla fattorizzazione di interi di grandi dimensioni, anche se dal punto di vista della matematica teorica, esiste la possibilità che, tramite la conoscenza della chiave pubblica, si possa decifrare il messaggio.

Sin dal brevetto, arrivato 6 anni dopo la pubblicazione del progetto, l'algoritmo RSA costituisce la base dei sistemi crittografici su cui si fondano i sistemi di sicurezza informatici utilizzati sulla rete Internet per l'autenticazione degli utenti. Attualmente, l'algoritmo è utilizzato per la cifratura delle firme digitali.

1.1 Il metodo

Per criptare un messaggio M con l'algoritmo RSA, usando la chiave pubblica $(e; n)$, è necessario rappresentare il messaggio come un numero intero compreso tra 0 e $n-1$. Se necessario, suddividere il messaggio in una serie di blocchi e rappresentare ogni blocco come intero con le caratteristiche richieste. Utilizzare una qualsiasi rappresentazione standard; in questa fase lo scopo non è di crittografare il messaggio ma solo di ottenerlo nella forma numerica necessaria per la crittografia. Quindi crittografare il messaggio elevandolo alla e-esima potenza. Il testo criptato C è dato dal resto della divisione, quando M^e è diviso per n.

Per decrittare il testo cifrato C, è sufficiente elevarlo alla d-esima potenza e salvare il resto della divisione per n.

Gli algoritmi di crittografia e di decrittografia E e D sono quindi:

$$C \equiv E(M) \equiv M^e \pmod{n} \text{ per un messaggio } M.$$
$$D(C) \equiv C^d \pmod{n} \text{ per il testo cifrato } C.$$

È importante notare che sia il messaggio originale M che testo criptato C sono numeri interi compresi tra 0 e $n-1$. Ciò vuol dire che l'algoritmo $E(M)$ non aumenta la dimensione del messaggio.

1.2 Come generare le chiavi

Il primo passo per generare le chiavi necessarie per questo algoritmo è di calcolare il prodotto n tra due numeri primi casuali p e q:

$$n = p \cdot q$$

Nonostante n venga resa pubblica, i fattori p e q saranno effettivamente nascosti agli occhi degli estranei a causa dell'estrema difficoltà nella fattorializzazione di n. Questo impedisce, inoltre, di ricavare d a partire da e.

Quindi si sceglie un intero d casuale, tale che sia relativamente primo con il prodotto tra i fattori $p-1$ e $q-1$, ovvero che soddisfi

$$\gcd(d, (p-1)\cdot(q-1)) = 1$$

Infine viene ricavato un intero e in modo tale che sia il "moltiplicativo inverso" di d, modulo $(p-1)\cdot(q-1)$:

$$e \cdot d \equiv 1 \pmod{(p-1)\cdot(q-1)}$$

Nei prossimi capitoli avremo modo di approfondire le reali procedure che servono per arrivare a generare le chiavi necessarie per il corretto svolgimento dell'algoritmo RSA.

1.3 La matematica retrostante

Dimostriamo la correttezza dell'algoritmo di decifrazione usando il teorema di Eulero: per un qualsiasi numero intero M, relativamente primo con n,

$$M^{\varphi(n)} \equiv 1 \pmod{n} \tag{1.1}$$

La funzione $\varphi(n)$ di Eulero, o *toziente*, è una funzione definita, per ogni intero positivo n, come il numero degli interi compresi tra 1 e n che sono coprimi con n. Una delle sue proprietà, che ci risulta molto utile, è che se p è un numero primo, allora

$$\varphi(p) = p - 1 \tag{1.2}$$

Nel nostro caso, abbiamo, per la proprietà fondamentale della funzione toziente

$$\begin{aligned} \varphi(n) &= \varphi(p) \cdot \varphi(q) \\ &= (p-1) \cdot (q-1) \end{aligned} \tag{1.3}$$

Poiché d è relativamente primo con $\varphi(n)$, ha un inverso moltiplicativo e nel cerchio degli interi modulo $\varphi(n)$

$$e \cdot d \equiv 1 \pmod{\varphi(n)} \tag{1.4}$$

Dato un messaggio M, il testo cifrato C viene calcolato con

$$C \equiv E(M) \equiv M^e \pmod{n}$$

Sostituendo, con l'algoritmo di decodifica, si ottiene

$$\begin{aligned} D(C) &\equiv C^d \pmod{n} \\ &\equiv (M^e)^d \pmod{n} \\ &\equiv M^{e \cdot d} \pmod{n} \end{aligned} \tag{1.5}$$

Dobbiamo dimostrare che tale valore è congruente al messaggio originale M. I valori e e d sono stati scelti in modo tale che l'equazione (1.4) fosse soddisfatta. Ovvero, deve esistere un numero $k \in \mathbb{Z}$ tale che

$$e \cdot d \equiv k \cdot \varphi(n) + 1 \tag{1.6}$$

Sostituendo in (1.5), si ottiene

$$\begin{aligned} M^{e \cdot d} &\equiv M^{k \cdot \varphi(n) + 1} \pmod{n} \\ &\equiv M^{k \cdot \varphi(n)} \cdot M \pmod{n} \\ &\equiv \left(M^{\varphi(n)}\right)^k \cdot M \pmod{n} \\ &\equiv (1)^k \cdot M \pmod{n} \\ &\equiv M \pmod{n} \end{aligned}$$

1.4 Un esempio pratico

Per fare una esempio pratico, se l'*utente A* vuole spedire un messaggio all'*utente B* e vuole essere sicuro che nessuno possa leggerlo, *A* dovrà chiedere a *B* di generare una coppia di chiavi e di rendere pubblica la chiave con la quale verrà cifrato il messaggio. Essendo l'*utente B* l'unico a possedere la chiave privata, sarà anche l'unico a poter decifrare il messaggio. In questo modo il testo sarà decifrabile esclusivamente dall'*utente B*, mentre chiunque visualizzi il messaggio non sarà in grado di comprenderlo.

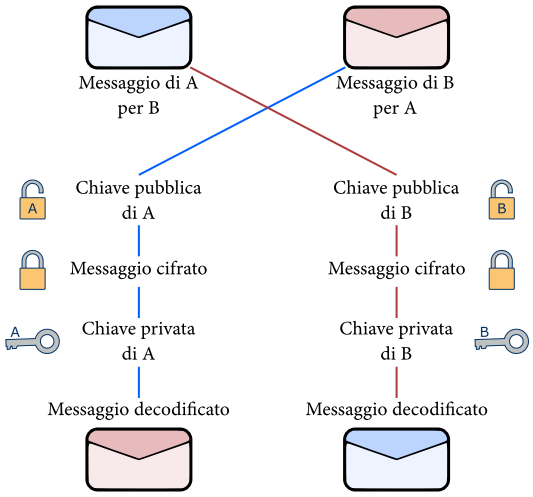


Figura 1.1: Esempio di corrispondenza elettronica utilizzando un sistema di crittografia asimmetrica.

Con questo metodo di cifratura è possibile anche garantire la provenienza di un messaggio. Riprendiamo l'esempio precedente: l'*utente A* questa volta, prima di cifrare il messaggio usando la chiave pubblica dell'*utente B*, lo cifrerà usando la propria chiave privata e solo in un secondo momento lo ri-crittograferà utilizzando la chiave pubblica di *B*. Quando l'*utente B* riceverà il messaggio e lo decifrerà usando la propria chiave inversa, otterrà ancora un messaggio crittografato. Quel dato messaggio necessiterà poi della chiave pubblica dell'*utente A* per essere decifrato, garantendo in questo modo che il messaggio è stato spedito soltanto dall'*utente A*, unico a possedere la chiave privata con la quale era stato crittografato il messaggio.

Più semplicemente, utilizzando questo metodo di cifratura, l'*utente A* può mandare messaggi a tutti, garantendo la provenienza. Infatti, cifrando il messaggio con la propria chiave privata, chiunque sarà in grado di leggere il messaggio, decifrandolo con la sua chiave pubblica, assicurandosi in tal modo che il mittente sia proprio l'*utente A*.

2
Aritmetica a precisione multipla

Una delle prime cose da fare per implementare da zero un sistema complesso come può essere un generatore di chiavi criptate, è sicuramente adattare gli algoritmi basilari che servono ad interagire con i numeri alle esigenze del software che si va a creare.

Questa implementazione si rende necessaria in quanto il tipo di dato in cui memorizzare un numero intero è di 32 *bit* di lunghezza, mentre il sistema andrà ad interagire con numeri le cui dimensioni variano in un *range* di *bit* molto elevato. L'attuale standard definito dal NIST[1] raccomanda un uso con chiavi lunghe 2048 *bit*, anche se è previsto un innalzamento dei sistemi di sicurezza con chiavi ancora più lunghe entro il prossimo decennio.

Anni	Dimensioni della chiave
Fino al 2010	1024 bit
Fino al 2030	2048 bit
Oltre il 2030	3072 bit

Tabella 2.1: Dimensioni consigliate delle chiavi RSA negli anni.

Gli algoritmi che devono essere adattati non si limitano esclusivamente alle quattro operazioni fondamentali dell'aritmetica, ma comprendono anche altre funzioni che fanno loro da corona e che li completano. Tra queste operazioni troviamo gli algoritmi di inizializzazione e di stampa, le operazioni *bit a bit*, gli algoritmi di confronto e quelli di incremento e di decremento unitario.

Per poter sopperire alla mancanza di strutture adeguate a memorizzare chiavi della lunghezza desiderata, e per poter effetturare le operazioni per il calcolo e la creazione delle stesse, è stata ideata una struttura composta da: un numero intero che descrive il segno del numero stesso e da un array di interi a 32 *bit* in cui viene salvato il modulo del numero.

Al fine rendere il più elastico possibile il software, nelle prime righe del codice sorgente, sono inserite delle definizioni che facilitano la modifica dei dati di base, quali la lunghezza delle chiavi, il numero di celle dell'array e il tipo di dato, senza segno, che compone l'array. Viene inoltre definito un ulteriore tipo di dato di lunghezza doppia rispetto al tipo di dato principale. Ad esempio, a fronte di un tipo di dato principale **uint32_t**, il dato di lunghezza doppia è **uint64_t**.

Per l'implementazione della matematica di base, sono stati implementati gli algoritmi descritti da Donald Knuth nel suo volume "The art of computer programming" [3]. Sebbene la loro complessità computazionale non sia tra le migliori, sono stati scelti per la forte adattabilità al problema posto. Infatti, tali algoritmi operano con un sistema numerico che può differire da quello decimale. In questo caso, gli algoritmi sono stati adattati alla struttura ad *array* illustrata in precedenza: al posto del sistema decimale, è stato impostato un sistema 2^{32}-esimale.

2.0.1 Algoritmi accessori

```
/*
 * Number of bit per BIG_NUMBER
 * Warning! Set a length twice as long as necessary.
 */
#define BIT_SIZE    4096

/*
 * Data type of word and extended word
 */
typedef uint32_t WORD_TYPE;
typedef uint64_t EXTWORD_TYPE;

/*
 * Size of word: number of bit per word
 */
#define WORD_SIZE    ( 8 * sizeof( WORD_TYPE ) )

/*
 * Number of word per BIG_NUMBER
 */
#define N_WORD       ( BIT_SIZE / WORD_SIZE )
```

```
/*
 * Max value of word
 */
#define WORD_MAX_VAL    0xFFFFFFFF



/* Data-holding structure: array of int
 *
 * Sign
 *   0 -> positive;
 *   1 -> negative;
 */
struct bn {
        char sign;
        WORD_TYPE num[N_WORD];
};



void bignum_init(struct bn* n) {
    int i;

    n->sign = 0;
    for (i = 0; i < N_WORD; i++) {
        n->num[i] = 0;
    }
}
```

2.1 Addizione

Dati due numeri interi non negativi di lunghezza n, rappresentati nella forma vettoriale

$$\vec{u} \leftarrow (u_{n-1}, \ldots, u_1, u_0)_b$$
$$\vec{v} \leftarrow (v_{n-1}, \ldots, v_1, v_0)_b$$

questo algoritmo restituisce la loro somma in base b,

$$\vec{w} \leftarrow (w_n, w_{n-1}, \ldots, w_1, w_0)_b$$

Il termine w_n costituisce il riporto dell'operazione. Il suo valore è sempre uguale a 0 oppure a 1, a seconda della presenza o meno del riporto.

Nel caso in cui uno dei due vettori abbia lunghezza $m < n$, è sufficiente aggiungere degli zeri per colmare la disparità venutasi a creare.

2.1.1 Pseudo-codice e complessità computazionale

Algoritmo 1: uAdd

Input: $\vec{u}, \vec{v}$
Output: $\vec{w} \leftarrow \vec{u} + \vec{v}$

```
1 begin
2     k ← 0
3     for j ← 0 to n do
4         w_j ← (u_j + v_j + k) mod b
5         k ← ⌊(u_j + v_j + k)/b⌋
6     w_n ← k
7     return w
```

```
void bignum_uadd(struct bn* u, struct bn* v, struct bn* w) {
    size_t j;
    EXTWORD_TYPE res = 0;

    for (j = 0; j < N_WORD; j++) {
        res = (EXTWORD_TYPE)u->num[j] +
              + (EXTWORD_TYPE)v->num[j] + res;

        w->num[j] = res;
        res >>= WORD_SIZE;
    }
    w->sign = 0;
}
```

È facile notare che la complessità computazionale di questo algoritmo è pari a $O(n)$.

2.2 Sottrazione

Dati due numeri interi non negativi di lunghezza n, rappresentati nella forma vettoriale

$$\vec{u} \leftarrow (u_{n-1}, \ldots, u_1, u_0)_b$$
$$\vec{v} \leftarrow (v_{n-1}, \ldots, v_1, v_0)_b$$

e tali che $\vec{u} > \vec{v}$, questo algoritmo restituisce la loro sottrazione in base b,

$$\vec{w} \leftarrow (w_{n-1}, \ldots, w_1, w_0)_b$$

Nel caso in cui $\vec{v}$ abbia lunghezza $m \leq n$, è sufficiente aggiungere

$$\forall j \in \{m, \ldots, n-1\}, v_j = 0$$

2.2.1 Pseudo-codice e complessità computazionale

Algoritmo 2: uSub

Input: $\vec{u}, \vec{v}$
Output: $\vec{w} \leftarrow \vec{u} - \vec{v}$

```
1 begin
2     k ← 0
3     for j ← 0 to n do
4         w_j ← (u_j − v_j − k) mod b
5         k ← ⌊(u_j − v_j − k) /b⌋
6     return w
```

```
void bignum_usub(struct bn* u, struct bn* v, struct bn* w) {
    size_t j;
    EXTWORD_TYPE k = 0;

    for (j = 0; j < N_WORD; j++) {
        EXTWORD_TYPE res = (EXTWORD_TYPE)u->num[j] +
              - (EXTWORD_TYPE)v->num[j] - k;

        w->num[j] = res;
        k = (res >> WORD_SIZE) ? 1 : 0;
    }
    w->sign = 0;
}
```

È importante notare che, $\forall j \in \{0, \ldots, n-1\}$, k è pari a 0 se nel calcolo di w_j non c'è stato riporto, 1 altrimenti.

Come per il precedente algoritmo di addizione, la complessità computazionale per l'algoritmo di sottrazione è $O(n)$.

2.3 Moltiplicazione

Dati due numeri interi non negativi rispettivamente di lunghezza m e n, rappresentati nella forma vettoriale

$$\vec{u} \leftarrow (u_{m-1}, \ldots, u_1, u_0)_b$$

$$\vec{v} \leftarrow (v_{n-1}, \ldots, v_1, v_0)_b$$

questo algoritmo restituisce la loro moltiplicazione in base b,

$$\vec{w} \leftarrow (w_{m+n-1}, \ldots, w_1, w_0)_b$$

2.3.1 Pseudo-codice e complessità computazionale

Algoritmo 3: uMul

Input: $\vec{u}, \vec{v}$
Output: $\vec{w} \leftarrow \vec{u} \cdot \vec{v}$

```
1  begin
2      for j ← 0 to n do
3          if (v_j = 0) then
4              w_{j+m} ← 0
5          else
6              k ← 0
7              for i ← 0 to m do
8                  t ← u_i · v_j + w_{i+j} + k
9                  w_{i+j} ← t mod b
10                 k ← ⌊t/b⌋
11             w_{j+m} ← k
12     return w
```

```
void bignum_umul(struct bn* u, struct bn* v, struct bn* w) {
    int m = bignum_sizeof(u);
    int n = bignum_sizeof(v);

    // Special case
    if ( n == 0 ) {
        bignum_init(w);
        return;
    }
    if ( n == 1 ) {
        bignum_umul_short(u, v->num[0], w);
```

```
        return;
    }

    size_t i, j;
    EXTWORD_TYPE res;
    bignum_init(w);

    for (j = 0; j < n; j++) {
        if ( v->num[j] == 0 ) {
            w->num[j + m] = 0;
        } else {
            res = 0;
            for (i = 0; i < m; i++) {
                res = ( (EXTWORD_TYPE)u->num[i] +
                      * (EXTWORD_TYPE)v->num[j] ) +
                      + (EXTWORD_TYPE)w->num[i + j] + res;
                w->num[i + j] = res;
                res >>= WORD_SIZE;
            }
            w->num[j + m] = res;
        }
    }
}
```

È importante inizializzare a zero il vettore risultante $\vec{w}$ prima di iniziare l'algoritmo, per evitare che operazioni quali

$$(w_{m+n-1}, \ldots, w_0)_b \leftarrow (u_{m-1}, \ldots, u_0)_b \cdot (v_{n-1}, \ldots, v_0)_b + (w_{m-1}, \ldots, w_0)_b$$

possano restituire valori errati a causa di una precedente occupazione dello spazio di memoria.

La complessità computazionale di questo algoritmo, al caso peggiore, è $O(m \cdot n)$, che si semplifica a $O(n^2)$ nel caso in cui entrambi i vettori abbiano la stessa lunghezza. Sebbene questo algoritmo sia molto semplice, non incarna la strada più rapida per effettuare la moltiplicazione tra i vettori $\vec{u}$ e $\vec{v}$ quando le loro lunghezze m e n sono grandi. Un procedimento computazionalmente meno oneroso può essere rappresentato dall'algoritmo di Karatsuba, dove la complessità cala del 27% circa. Di primo acchito, l'implementazione dell'ultima procedura può sembrare la migliore, ma dopo una breve analisi ci si rende conto che l'iniziale vantaggio dato dalla complessità migliore, svanisce e arretra a causa del crescente spazio di memoria occupato. Come descritto brevemente nell'introduzione, in un dispositivo *embedded* è necessario trovare un giusto compromesso tra complessità computazionale e occupazione di memoria.

2.3.2 Pseudo-codice per una "moltiplicazione corta"

Nei casi in cui il vettore $\vec{v}$ abbia lunghezza unitaria, viene utilizzato l'algoritmo mostrato di seguito. Tale procedura ha una complessità computazionale lineare, che la rende preferibile all'algoritmo 3.

Algoritmo 4: uMul-short

Input: $\vec{u}, v$
Output: $\vec{w} \leftarrow \vec{u} \cdot v$

```
1 begin
2     for j ← 0 to n do
3         Spostare il vettore w a sinistra di una parola.
4         w_0 ← ⌊(u_j · v + k) / b⌋
5         k ← (u_j × v + k) mod b
6     w_j ← k
7     return w
```

```
void bignum_umul_short(struct bn* u, WORD_TYPE v, struct bn* w)
{
    bignum_init(w);

    int i, uLength = bignum_sizeof(u);

    EXTWORD_TYPE multiplicand = 0;
    EXTWORD_TYPE carry = 0;
    for (i = 0; i < uLength; i++) {
        multiplicand = ( (EXTWORD_TYPE)u->num[i] +
              * (EXTWORD_TYPE)v ) + carry;
        w->num[i] = multiplicand;
        carry = multiplicand >> WORD_SIZE;
    }

    w->num[uLength] = carry;
}
```

2.4 Divisione

Dati due numeri interi non negativi rispettivamente di lunghezza $m+n$ e n, rappresentati nella forma vettoriale

$$\vec{u} \leftarrow (u_{m+n-1}, \ldots, u_1, u_0)_b$$

$$\vec{v} \leftarrow (v_{n-1}, \ldots, v_1, v_0)_b$$

dove $v_{n-1} \neq 0$, questo algoritmo restituisce quoziente e resto, espressi in base b,

$$\vec{q} \leftarrow \lfloor \vec{u}/\vec{v} \rfloor \leftarrow (q_m, q_{m-1}, \ldots, q_1, q_0)_b$$

$$\vec{r} \leftarrow \vec{u} \bmod \vec{v} \leftarrow (r_{n-1}, \ldots, r_1, r_0)_b$$

2.4.1 Pseudo-codice e complessità computazionale

Algoritmo 5: uDiv

Input: $\vec{u}, \vec{v}$
Output: $\vec{q} \leftarrow \lfloor \vec{u}/\vec{v} \rfloor$
$\vec{r} \leftarrow \vec{u} \bmod \vec{v}$

1 **begin**
2 Spostare il vettore $\vec{v}$ di d posizioni verso sinistra, affinché il bit più significativo di v_{n-1} sia 1.
3 Spostare il vettore $\vec{u}$ di d posizioni verso sinistra.
4 **for** $j \leftarrow m$ **down to** 0 **do**
5 $\hat{q} \leftarrow \lfloor \left(u_{j+n} b + u_{j+n-1}\right) / v_{n-1} \rfloor$
6 $\hat{r} \leftarrow \left(u_{j+n} b + u_{j+n-1}\right) \bmod v_{n-1}$
7 **if** $(\hat{q} = b)$ *or* $(\hat{q} v_{n-2} > b\hat{r} + u_{j+n-2})$ **then**
8 $\hat{q} \leftarrow \hat{q} - 1$
9 $\hat{r} \leftarrow \hat{r} + v_{n-1}$
10 Ripetere questo test finché $\hat{r} < b$
11 Sostituire $\left(u_{j+n}, u_{j+n-1}, \ldots, u_j\right)_b$ con $\left(u_{j+n}, u_{j+n-1}, \ldots, u_j\right)_b - \hat{q} \cdot (v_{n-1}, \ldots, v_1, v_0)_b$
12 $q_j \leftarrow \hat{q}$
13 **while** $|\vec{u}| < 0$ **do**
14 $q_j \leftarrow q_j - 1$
15 $\left(u_{j+n}, \ldots, u_j\right)_b \leftarrow \left(u_{j+n}, \ldots, u_j\right)_b + (v_{n-1}, \ldots, v_0)_b$
16 Spostare il vettore $\vec{u}$ di d posizioni verso destra, salvandolo in $\vec{r}$.
17 **return** $\vec{q}, \vec{r}$

```
void bignum_udiv(struct bn* a, struct bn* b,
    struct bn* quotient, struct bn* remainder) {
    int m = bignum_sizeof(a);
    int n = bignum_sizeof(b);

    if ( n == 0 ) {
        perror("Error: impossible divide by 0.");
        return 1;
    }
    if ( n == 1 ) {
        bignum_udiv_short(a, b->num[0], quotient, remainder);
        return;
    }

    bignum_init(quotient);
    bignum_init(remainder);

    if ( m < n ) {
        // q = 0, r = a
        bignum_assign(remainder, a);
        return;
    }

    if ( m == n ) {
        int cmp = bignum_cmp(a, b);

        if ( cmp == SMALLER ) {
            // q = 0, r = a
            bignum_assign(remainder, a);
            return;
        }
        if ( cmp == EQUAL ) {
            // q = 1, r = 0
            bignum_from_word(quotient, 0, 1);
            return;
        }
    }


    EXTWORD_TYPE base = (EXTWORD_TYPE)1 << WORD_SIZE;
    m = m - n;
```

```
    /*
     * Step D1: normalize
     */
    int divShift;
    for (divShift = WORD_SIZE; divShift > 0; divShift--) {
        if ( ((b->num[n - 1] >> (divShift - 1)) & 1) != 0 ) {
            break;
        }
    }
    divShift = WORD_SIZE - divShift;

    struct bn u, v;
    bignum_lshift(a, &u, divShift);
    bignum_lshift(b, &v, divShift);


    /*
     * Step D2: Initialize j.
     */
    int j;

    for (j = m; j >= 0; j--) {
        /*
         * Step D3: Calculate Q'.
         */
        EXTWORD_TYPE product = ((EXTWORD_TYPE)u.num[j + n] <<
              << WORD_SIZE) + (EXTWORD_TYPE)u.num[j + n - 1];
        EXTWORD_TYPE qh = product / (EXTWORD_TYPE)v.num[n - 1];
        EXTWORD_TYPE rh = product % (EXTWORD_TYPE)v.num[n - 1];

        EXTWORD_TYPE leftHand, rightHand;
        char flag = FALSE;

        do {
            leftHand = qh * (EXTWORD_TYPE)v.num[n - 2];
            rightHand = (rh << WORD_SIZE) +
                  + (EXTWORD_TYPE)u.num[j + n - 2];

            if ( (qh == base) || (leftHand > rightHand) ) {
                qh--;
                rh = rh + (EXTWORD_TYPE)v.num[n - 1];
                flag = TRUE;
            } else {
```

```
                flag = FALSE;
            }
        } while ( flag && (rh < base) );


        /*
         * Step D4 & D6: Multiply and subtract & add back.
         */
        struct bn vv, tmp;
        qh++;

        do {
            qh--;
            bignum_lshift(&v, &tmp, j * WORD_SIZE);
            bignum_umul_short(&tmp, (WORD_TYPE)qh, &vv);
        } while ( bignum_cmp(&u, &vv) == SMALLER );

        bignum_usub(&u, &vv, &tmp);
        bignum_assign(&u, &tmp);


        /*
         * Step D5: Test remainder.
         */
        quotient->num[j] = qh;


        /*
         * Step D7: Loop on j.
         */
    }


    /*
     * Step D8: Unnormalize.
     */
    bignum_rshift(&u, remainder, divShift);
    return;
}
```

Come nel caso della moltiplicazione, la divisione può essere fatta, fondamentalmente, con il metodo tradizionale della scuola elementare. I dettagli, tuttavia, sono sorprendentemente complicati. Anche per questo algoritmo la complessità computazionale è quadratica, derivata principalmente dalla moltiplicazione, utilizzato ad ogni iterazione

del ciclo *for*.

2.4.2 Pseudo-codice per "divisione corta"

Nei casi in cui il vettore $\vec{v}$ abbia lunghezza unitaria, viene utilizzato l'algoritmo mostrato di seguito. Tale procedura ha una complessità computazionale lineare, che la rende preferibile all'algoritmo 5.

Algoritmo 6: uDiv-short

Input: $\vec{u}, v$
Output: $\vec{q} \leftarrow \lfloor \vec{u}/v \rfloor$
$\vec{r} \leftarrow \vec{u} \bmod v$

1 begin
2 for $j \leftarrow n-1$ down to 0 do
3 Spostare il vettore $\vec{q}$ a sinistra di una parola.
4 $q_0 \leftarrow \lfloor \left(k \cdot b + u_j\right)/v \rfloor$
5 $k \leftarrow \left(k \cdot b + u_j\right) \bmod v$
6 $r_0 \leftarrow k$
7 return $\vec{q}, \vec{r}$

```
void bignum_udiv_short(struct bn* u, WORD_TYPE v,
    struct bn* q, struct bn* r) {
    bignum_init(q);
    bignum_init(r);

    int i, aLength = bignum_sizeof(u);

    EXTWORD_TYPE dividend = 0;
    WORD_TYPE rem = 0;
    for (i = aLength; i >= 0; i--) {
        dividend = ((EXTWORD_TYPE)rem << WORD_SIZE) + u->num[i];

        _lshift_word(q, 1);
        q->num[0] = dividend / (EXTWORD_TYPE)v;
        rem = dividend % (EXTWORD_TYPE)v;
    }

    bignum_from_word(r, 0, rem);
}
```

2.5 Operazioni matematiche con segno

Negli algoritmi visti fino ad ora, non sono stati presi in considerazione i segni degli operandi. In questo modo le operazioni svolte erano esclusivamente positive.

Per ovviare a questo problema, sono stati introdotti degli ulteriori algoritmi che effettuano dei controlli sugli operandi e sui loro segni, richiamano i rispettivi algoritmi *unisgned* e calcolano il segno del risultato.

2.5.1 Addizione

Per effettuare un'addizione con segno, è necessario distinguere 4 casi, in base al segno degli operandi:

1. $W = (+U) + (+V)$. In questo caso il modulo del risultato è la mera somma degli operandi; il segno è positivo.

2. $W = (+U) + (-V)$. L'equazione può essere riscritta cambiando il segno del secondo operando e dell'operazione stessa, diventando una sottrazione $W = (+U) - (+V)$. Si distinguono due ulteriori casi:

   - Se $|U| \geq |V|$, allora $W = (+U) - (+V)$ con segno positivo.
   - Se $|U| < |V|$, allora $W = (+V) - (+U)$ con segno negativo.

3. $W = (-U) + (+V)$. L'equazione può essere riscritta invertendo i fattori, cambiando il segno del primo operando e dell'operazione stessa, diventando una sottrazione $W = (+V) - (+U)$. Si distinguono due ulteriori casi, speculari al secondo caso:

   - Se $|V| \geq |U|$, allora $W = (+V) - (+U)$ con segno positivo.
   - Se $|V| < |U|$, allora $W = (+U) - (+V)$ con segno negativo.

4. $W = (-U) + (-V)$. In modo speculare al primo caso, il modulo del risultato è la mera somma degli operandi, mentre il segno è negativo.

Algoritmo 7: add

Input: $\vec{u}, \vec{v}$
Output: $\vec{w} \leftarrow \vec{u} + \vec{v}$

```
1  begin
       /* FIRST and FOURTH CASES                                    */
2      if u.sign() = v.sign() then
3          |w| ← uAdd(u, v)
4          w.sign() ← u.sign()
       /* SECOND CASE                                               */
5      if u.sign() = 0 and v.sign() = 1 then
6          if |u| > |v| then
7              |w| ← uSub(u, v)
8              w.sign() ← 0
9          else
10             |w| ← uSub(v, u)
11             w.sign() ← 1
       /* THIRD CASE                                                */
12     if u.sign() = 1 and v.sign() = 0 then
13         if |u| > |v| then
14             |w| ← uSub(u, v)
15             w.sign() ← 1
16         else
17             |w| ← uSub(v, u)
18             w.sign() ← 0
19     return w
```

```
void bignum_add(struct bn* a, struct bn* b, struct bn* c) {
    // FIRST  and  FOURTH  CASES
    if ( a->sign == b->sign ) {
        bignum_uadd(a, b, c);
        c->sign = a->sign;
        return;
    }

    // SECOND  CASE:  A - B
    if ( ( a->sign == 0 ) && ( b->sign == 1 ) ) {
        if ( bignum_cmp(a, b) != SMALLER ) {
            bignum_usub(a, b, c);
            c->sign = 0;
```

```
        } else {
            bignum_usub(b, a, c);
            c->sign = 1;
        }
    return;
    }

    // THIRD  CASE:  B - A
    if ( ( a->sign == 1 ) && ( b->sign == 0 ) ) {
        if ( bignum_cmp(b, a) != SMALLER ) {
            bignum_usub(b, a, c);
            c->sign = 0;
        } else {
            bignum_usub(a, b, c);
            c->sign = 1;
        }
    return;
    }
}
```

2.5.2 Sottrazione

Per effettuare una sottrazione con segno, è necessario distinguere 4 casi, in base al segno degli operandi:

1. $W = (+U) - (+V)$. È necessario distinguere due ulteriori casi:
   - Se $|U| \geq |V|$, allora $W = (+U) - (+V)$ con segno positivo.
   - Se $|U| < |V|$, allora $W = (+V) - (+U)$ con segno negativo.
2. $W = (+U) - (-V)$. L'equazione può essere riscritta cambiando il segno del secondo operando e dell'operazione stessa, diventando una somma $W = (+U) + (+V)$. Il segno è positivo.
3. $W = (-U) - (+V)$. L'equazione può essere riscritta come $W = -((+U) + (+V))$, quindi il modulo è la somma degli operatori, mentre il segno è negativo.
4. $W = (-U) - (-V)$. Dopo le opportune modifiche, l'equazione diventa $W = (+V) - (+U)$. Specularmente al primo caso si individuano due ulteriori casi:
   - Se $|V| \geq |U|$, allora $W = (+V) - (+U)$ con segno positivo.
   - Se $|V| < |U|$, allora $W = (+U) - (+V)$ con segno negativo.

Algoritmo 8: sub

Input: $\vec{u}, \vec{v}$
Output: $\vec{w} \leftarrow \vec{u} + \vec{v}$

```
1  begin
       /* FIRST CASE                                                    */
2      if u.sign() = 0 and v.sign() = 0 then
3          if |u| > |v| then
4              |w| ← uSub(u, v)
5              w.sign() ← 0
6          else
7              |w| ← uSub(v, u)
8              w.sign() ← 1
       /* SECOND and THIRD CASES                                        */
9      if u.sign() ≠ v.sign() then
10         |w| ← uAdd(u, v)
11         w.sign() ← u.sign()
       /* FOURTH CASE                                                   */
12     if u.sign() = 1 and v.sign() = 0 then
13         if |u| > |v| then
14             |w| ← uSub(u, v)
15             w.sign() ← 1
16         else
17             |w| ← uSub(v, u)
18             w.sign() ← 0
19     return w
```

```
void bignum_sub(struct bn* a, struct bn* b, struct bn* c) {
    // FIRST  CASE:  A - B
    if ( ( a->sign == 0 ) && ( b->sign == 0 ) ) {
        if ( bignum_cmp(a, b) != SMALLER ) {
            bignum_usub(a, b, c);
            c->sign = 0;
        } else {
            bignum_usub(b, a, c);
            c->sign = 1;
        }
    return;
    }
```

```
    // SECOND  and  THIRD  CASES
    if ( a->sign != b->sign ) {
        bignum_uadd(a, b, c);
        c->sign = a->sign;
        return;
    }

    // FOURTH  CASE:  B - A
    if ( ( a->sign == 1 ) && ( b->sign == 1 ) ) {
        if ( bignum_cmp(b, a) != SMALLER ) {
            bignum_usub(b, a, c);
            c->sign = 0;
        } else {
            bignum_usub(a, b, c);
            c->sign = 1;
        }
    return;
    }
}
```

2.5.3 Moltiplicazione

Per la moltiplicazione, il calcolo del segno viene effettuato tramite una semplice operazione di *XOR* tra i segni degli operandi.

Algoritmo 9: mul

Input: $\vec{u}, \vec{v}$
Output: $\vec{w} \leftarrow \vec{u} \cdot \vec{v}$

1 begin
2 $\quad |\vec{w}| \leftarrow \text{uMul}(\vec{u}, \vec{v})$
3 $\quad \vec{w}.sign() \leftarrow \vec{u}.sign() \oplus \vec{v}.sign()$
4 $\quad$ return $\vec{w}$

```
void bignum_mul(struct bn* a, struct bn* b, struct bn* c) {

    bignum_umul(a, b, c);
    c->sign = a->sign ^ b->sign;
}
```

2.5.4 Divisione e Modulo

Pari modo alla moltiplicazione, avviene per la divisione. L'unica differenza è che il vettore resto $\vec{r}$, uno dei due risultati dell'algoritmo *uDiv*, viene tenuto in considerazione esclusivamente dall'algoritmo 11.

Algoritmo 10: div

Input: $\vec{u}, \vec{v}$
Output: $\vec{q} \leftarrow \lfloor \vec{u}/\vec{v} \rfloor$

1 begin
2 $(|\vec{q}|, |\vec{r}|) \leftarrow \text{uDiv}(\vec{u}, \vec{v})$
3 $\vec{q}.sign() \leftarrow \vec{u}.sign() \oplus \vec{v}.sign()$
4 return $\vec{q}$

```
void bignum_div(struct bn* a, struct bn* b, struct bn* c) {
    struct bn tmp;

    bignum_udiv(a, b, c, &tmp);
    c->sign = a->sign ^ b->sign;
}
```

Algoritmo 11: mod

Input: $\vec{u}, \vec{v}$
Output: $\vec{r} \leftarrow \vec{u} \bmod \vec{v}$

1 begin
2 $(|\vec{q}|, |\vec{r}|) \leftarrow \text{uDiv}(\vec{u}, \vec{v})$
3 $\vec{r}.sign() \leftarrow 0$
4 return $\vec{r}$

```
void bignum_mod(struct bn* a, struct bn* b, struct bn* c) {
    struct bn tmp;

    bignum_udiv(a, b, &tmp, c);
    c->sign = 0;
}
```

3
Test di Primalità

Nell'implementazione di un crittosistema RSA è necessario generare due numeri primi molto grandi. In sostanza, vengono generati casualmente dei numeri interi dispari e viene testata la loro primalità grazie ad un procedimento sviluppato da *Robert M. Solovay* e *Volker Strassen*, un algoritmo probabilistico il cui tempo di esecuzione è polinomiale.

Per comprendere al meglio i passaggi che stanno dietro l'esecuzione del test, è necessario introdurre alcune nozioni di Teoria dei Numeri.

La prima definizione che si incontra è quella di *residuo quadratico*, un concetto fondamentale nei crittosistemi a chiave asimmetrica in quanto la loro teoria è basata sulla matematica modulare e sulle rispettive proprietà.

Definizione 3.1 (Residuo quadratico)**.** Un numero intero q è detto residuo quadratico modulo p se esiste un intero x tale che

$$x^2 \equiv q \pmod{p}$$

In caso contrario, q è detto residuo non quadratico.

A seguire, la definizione del *simbolo di Legendre*, che viene utilizzato per determinare se un numero è o meno primo.

Definizione 3.2 (Simbolo di Legendre). Siano p un numero primo ed a un intero, allora:

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{se } p \text{ divide } a \\ -1 & \text{se } a \text{ è un residuo quadratico modulo } p \\ 1 & \text{se } a \text{ non è un residuo quadratico modulo } p \end{cases}$$

$\left(\frac{a}{p}\right)$ è detto simbolo di Legendre.

Per poter effettuare il test, verrà però usato il *simbolo di Jacobi*, una generalizzazione del simbolo di Legendre, valida per un qualsiasi numero dispari.

Teorema 3.1 (Teorema di Eulero). Per il Simbolo di Legendre vale la seguente relazione:

$$\left(\frac{a}{p}\right) \equiv_p a^{\frac{p-1}{2}}$$

Dimostrazione. Se p divide a, l'asserto segue banalmente.
Supponiamo quindi che a sia un residuo quadratico modulo p. Troviamo un numero k tale che

$$k^2 \equiv a \pmod{p}$$

Allora, per il piccolo teorema di Fermat,

$$\begin{aligned} a^{(p-1)/2} &\equiv k^{(p-1)} \pmod{p} \\ &\equiv 1 \pmod{p} \end{aligned}$$

Viceversa, assumiamo che

$$a^{(p-1)/2} \equiv 1 \pmod{p}$$

Sia α un elemento primitivo modulo p, in modo tale che $a = \alpha^i$. Sostituendo, si ottiene:

$$\alpha^{i\cdot(p-1)/2} \equiv 1 \pmod{p}$$

Per il piccolo teorema di Fermat, $(p-1)$ divide $i \cdot (p-1)/2$, perciò i deve essere pari.
Sia $k \equiv \alpha^{i/2} \pmod{p}$. Abbiamo infine che

$$k^2 = \alpha^i \equiv a \pmod{p}$$

□

Il criterio di Eulero è correlato alla legge di reciprocità quadratica ed è utilizzato nella definizione degli pseudoprimi di Eulero-Jacobi.

Definizione 3.3 (Simbolo di Jacobi)**.** Siano n un numero intero dispari ed a un intero. Se $n = \prod_{i=1}^{k} p_i^{\alpha_i}$, $p_i \in \mathbb{P}$ e $\alpha_i \in \mathbb{N}$, allora si definisce Simbolo di Jacobi $\left(\frac{a}{n}\right)$ il prodotto dei Simboli di Legendre dei fattori primi di n:

$$\left(\frac{a}{p}\right) = \prod_{i=1}^{k} \left(\frac{a}{p_i}\right)^{\alpha_i}$$

Si noti che $\left(\frac{a}{n}\right) = 1$, con n composto, non implica che a sia un residuo quadratico modulo n: questo accade se e solo se a è residuo quadratico modulo ogni primo che divide n. Ne è un esempio $\left(\frac{2}{15}\right)$, dove il calcolo del Simbolo di Jacobi restituisce $\left(\frac{2}{15}\right) = \left(\frac{2}{3}\right) \cdot \left(\frac{2}{5}\right) = (-1) \cdot (-1) = 1$, mentre non esiste un intero x tale che $x^2 \equiv_{15} 2$.

Algoritmo 12: jacobi

Input: $\vec{a}, \vec{n}$
Output: $J \leftarrow \left(\frac{a}{n}\right)$

```
1  begin
2      if ā = 0 then
3          return 0
4      if ā = 1 then
5          return 1
           /* Scrivere ā come 2^e · Ā                                  */
6      if e ≡ 0 (mod 2) then
7          s ← 1
8      else
9          if n̄ ≡ 1 (mod 8) or n̄ ≡ 7 (mod 8) then
10             s ← +1
11         if n̄ ≡ 3 (mod 8) or n̄ ≡ 5 (mod 8) then
12             s ← −1
13     if n̄ ≡ 3 (mod 4) and Ā ≡ 3 (mod 4) then
14         s ← −s
15     N̄ ← n̄ mod Ā
16     if Ā = 1 then
17         return s
18     return s · jacobi(N̄, Ā)
```

```
int rsa_jac(struct bn* a, struct bn* n) {
    if ( bignum_is_zero(a) ) {
        return 0;
    }

    if ( bignum_cmp_short(a, 1) == EQUAL ) {
        return 1;
    }

    WORD_TYPE e = 0;
    struct bn a1, n1;
    bignum_assign(&a1, a);

    for (e = 0; !(a1.num[0] & 0x01); e++) {
        _rshift_one_bit(&a1);
    }

    int s;
    if ( e % 2 == 0 ) {
        s = 1;
    } else {
        if ( ((n->num[0] & 0x07) == 0x01) ||
             || ((n->num[0] & 0x07) == 0x07) ) {
            s = 1;
        } else {
        s = -1;
        }
    }

    if ( ((n->num[0] & 0x03) == 0x03) &&
          && ((a1.num[0] & 0x03) == 0x03) ) {
        s = -s;
    }

    if ( bignum_cmp_short(&a1, 1) != 0 ) {
        bignum_mod(n, &a1, &n1);
        s = s * rsa_jac(&n1, &a1);
    }

    return s;
}
```

3.1 Pseudoprimi

Definizione 3.4 (Pseoudoprimo di Eulero)**.** Siano n un intero composto dispari e b un intero tale che $gcd(b, n) = 1$. Allora n si dice pseudoprimo di Eulero rispetto alla base b se

$$b^{\frac{n-1}{2}} \equiv \left(\frac{b}{n}\right) \pmod{n}$$

dove $\left(\frac{b}{n}\right)$ è il simbolo di Jacobi.

Si noti che ogni numero primo n ($\neq 2$) soddisfa quest'ultima definizione in virtù del teorema 3.1.

Nella definizione di pseudoprimo di Eulero, si incontra la funzione $gcd(b, n)$, ovvero quella funzione che determina il *massimo comun divisore*. Per poterlo calcolare, si ricorre all'algoritmo di Euclide.

Algoritmo 13: gcd

```
Input: a⃗, b⃗
Output: c⃗ ← gcd(a⃗, b⃗)
1 begin
2     while b⃗ ≠ 0 do
3         r⃗ ← a⃗ mod b⃗
4         a⃗ ← b⃗
5         b⃗ ← r⃗
6     return a⃗
```

```
void rsa_gcd(struct bn* a, struct bn* b, struct bn* c) {
    struct bn newA, newB, tmp;

    bignum_assign(&newA, a);
    bignum_assign(&newB, b);

    while ( !bignum_is_zero(&newB) ) {
        bignum_mod(&newA, &newB, &tmp);
        bignum_assign(&newA, &newB);
        bignum_assign(&newB, &tmp);
    }

    bignum_assign(c, &newA);
}
```

L'algoritmo, con complessità polinomiale, serve a trovare il massimo comun divisore tra due numeri, che ricordiamo essere molto grandi (circa 1024 bits). L'innovazione data

dallo scopritore di questo procedimento[1] è il fatto di non trovare il risultato mediante fattorizzazione dei numeri, ma tramite il resto della loro divisione.

Troveremo, nei prossimi capitoli, una generalizzazione dell'*algoritmo di Euclide* quando tratteremo della generazione delle chiavi per la crittografia, in particolare per la generazione della chiave di decrittazione.

[1] Sebbene porti il nome di Euclide (IV secolo a.C. - III secolo a.C.), l'algoritmo era conosciuto anche nel 375 a.C. e probabilmente anche 200 anni prima.

3.2 Test di primalità

Un test di primalità è un algoritmo che ha lo scopo di determinare se un dato numero intero è primo. Non va confuso con gli algoritmi di fattorizzazione, che invece hanno lo scopo di determinare i fattori primi di un numero.

Attualmente, i test di primalità più efficienti sono probabilistici, nel senso che danno una risposta certa solo quando negano la primalità di un numero, mentre nel caso di risposte confermative, assicurano soltanto un limite inferiore alla probabilità che tale numero sia primo. La probabilità di errore dei test può essere però ridotta eseguendo l'algoritmo un certo numero di volte.

Tra le diverse procedure disponibili, per questo crittosistema è stato sviluppato l'algoritmo di Solovay-Strassen, sviluppato nel 1977. Essenzialmente, il test è basato sugli pseudoprimi di Eulero, visti nelle pagine precedenti.

Algoritmo 14: checkPrimality

Input: $\vec{n}$
Output: "n è primo." oppure "n è un intero composto."

1 **begin**
2 Si scelga un intero a compreso tra 1 e $n-1$
3 $x \leftarrow \left(\frac{a}{n}\right)$
4 **if** $x = 0$ **then**
5 **return** "n è *un intero composto.*"
6 $y \leftarrow a^{(n-1)/2} \bmod n$
7 **if** $x \equiv y \pmod{n}$ **then**
8 **return** "n è *primo.*"
9 **return** "n è *un intero composto.*"

```
int rsa_check_prime(struct bn* n) {
    struct bn a;
    rsa_random_under(&a, n);

    struct bn ret_gcd;
    rsa_gcd(n, &a, &ret_gcd);
    if ( bignum_cmp_short(&ret_gcd, 1) != EQUAL ) {
        return COMPOSITE;
    }

    struct bn ret_jac;
    switch ( rsa_jac(&a, n) ) {
        case -1:
            bignum_usub_short(n, 1, &ret_jac);
```

```
            break;

        case 0:
            return COMPOSITE;
            break;

        case 1:
            bignum_from_word(&ret_jac, 0, 1);
            break;

        default:
            return COMPOSITE;
            break;
    }

    struct bn ret_jac_eul, exp;
    bignum_usub_short(n, 1, &exp);
    _rshift_one_bit(&exp);
    rsa_pow_mod(&a, &exp, n, &ret_jac_eul);

    if ( bignum_cmp(&ret_jac, &ret_jac_eul) != EQUAL ) {
        return COMPOSITE;
    }
        return PRIME;
    }
}
```

Dall'analisi dell'algoritmo appena visto, segue che l'output "n è un intero composto" è sempre corretto, mentre la probabilità che l'output "n è primo" sia scorretto è minore o uguale a $1/2$.

È lecito, a questo punto, domandarsi quanti numeri interi casuali è necessario testare per trovarne uno che sia primo.

Teorema 3.2 (Teorema dei numeri primi)**.** Sia x un numero reale positivo e si definisca la funzione $\pi(x)$ come il numero di primi minori o uguali a x. Allora

$$\pi(x) \simeq \frac{x}{\ln x}$$

Pertanto la probabilità che un numero p sia primo è $1/\ln x$. Se p è di $1024bit$, come consigliato fino al 2030, allora la probabilità è pari a

$$\begin{aligned}\frac{1}{\ln 2^{1024}} &\simeq \frac{1}{\ln e^{710}} \\ &\simeq \frac{1}{710}\end{aligned}$$

In media, quindi, viene trovato un numero primo di lunghezza 1*kbit* ogni 710 interi casuali.

Tenendo conto, però, che i numeri testati sono esclusivamente dispari, tale probabilità viene raddoppiata, per cui è lecito supporre che venga trovato un numero primo ogni 355 numeri generati casualmente dall'algoritmo.

4
Generazione delle chiavi

La generazione della coppia di chiavi parte dalla scelta di due numeri primi casuali molto grandi, p e q. Essi vanno scelti con il testi di primalità visto nel capitolo precedente. Il motivo della richiesta che tali numeri siano molto grandi, risiede nella difficoltà di fattorizzare il loro prodotto $n = p \cdot q$, in quanto parte della chiave pubblica[1]. Come quanto detto nel capitolo 2, ad oggi la raccomandazione è di utilizzare chiavi di lunghezza $2kbit$. Per essere più precisi, la lunghezza della chiave si riferisce al prodotto n, per cui sostanzialmente, p e q devono essere lunghi la metà di n, ovvero $1kbit$.

Per ottimizzare la ricerca delle chiavi e e d, viene effettuato un procedimento diverso da quello illustrato nel secondo capitolo. Per prima cosa viene scelto un numero e primo. Non è necessario che tale numero sia grande. Solitamente viene scelto un numero appartenente al seguente insieme di numeri: $3, 5, 17, 257, 65537$. Questi particolari valori vengono scelti perché rendono più veloce l'operazione di esponenziazione modulare, avendo solo due bit di valore 1 nella loro rappesentazione binaria.

Tali numeri sono i primi cinque numeri di Fermat, indicati come $F_0, \ldots, F_4$, dove

$$F_k = 2^{2^k} + 1$$

Fermat credeva, erroneamente, che tutti i numeri della forma indicata fossero primi. Nel 1732, Eulero dimostrò che Fermat si sbagliava, fornendo la fattorizzazione di F_5:

$$F_5 = 4294967297 = 641 \cdot 6700417$$

[1] Si ricorda che la chiave pubblica è composta dalla coppia (n, e), mentre la coppia (n, d) identifica la chiave privata.

La scelta più usuale per e è $F_4 = 65537 = 0x10001$. Inoltre, avendo scelto e, è più semplice verificare le uguaglianze

$$\begin{cases} gcd(e, p-1) = 1 \\ gcd(e, q-1) = 1 \end{cases} \tag{4.1}$$

durante la generazione dei numeri primi. Nel caso in cui p o q falliscano questa verifica, possono essere rifiutati e la procedura viene ricominciata.

Viene quindi calcolata la funzione *toziente* (1.2),

$$\varphi(n) = (p-1) \cdot (q-1)$$

Si noti che la verifica del sistema (4.1) implica la verifica anche di

$$gcd(e, \varphi(n)) = 1$$

```
int generateRSAPrime(struct bn * p, size_t nbits, WORD_TYPE e,
    size_t ntests, const unsigned char *seed, size_t seedlen) {
    /* Create a prime p such that gcd(p-1, e) = 1.
     * Returns # prime tests carried out or -1 if failed.
     * Sets the TWO highest bits to ensure that the
     * product pq will always have its high bit set.
     * e MUST be a prime > 2.
     * This function assumes that e is prime so we can
     * do the less expensive test p mod e != 1 instead
     * of gcd(p-1, e) == 1.
     */

    struct bn u;
    size_t i, j, iloop, maxloops, maxodd;
    int done, overflow, failedtrial;
    int count = 0;
    WORD_TYPE r[N_SMALL_PRIMES];

    // Create a temp
    bignum_init(&u);

    maxodd = nbits * 100;
    maxloops = 5;

    done = 0;
    for (iloop = 0; !done && iloop < maxloops; iloop++) {
        // Set candidate n0 as random odd number
```

```
        rsa_random(p, nbits, seed);

        // Set two highest and low bits
        rsa_set_bit(p, nbits - 1);
        rsa_set_bit(p, nbits - 2);
        rsa_set_bit(p, 0);

        // To improve trial division, compute table
        // R[q] = n0 mod q for each odd prime q <= B
        for (i = 0; i < N_SMALL_PRIMES; i++) {
            bignum_mod_short(p, SMALL_PRIMES[i], &u);
            r[i] = u.num[0];
        }

        done = overflow = 0;
        // Try every odd number n0, n0+2, n0+4, ...
        // until we succeed
        for (j = 0; j < maxodd; j++, bignum_dbl_inc(p)) {
            count++;

            // Each time 2 is added to the current candidate
            // update table R[q] = (R[q] + 2) mod q
            if (j > 0) {
                for (i = 0; i < N_SMALL_PRIMES; i++) {
                    r[i] = (r[i] + 2) % SMALL_PRIMES[i];
                }
            }

            // Candidate passes the trial division stage if
            // and only if NONE of the R[q] values equal zero
            failedtrial = 0;
            for (i = 0; i < N_SMALL_PRIMES; i++) {
                if (r[i] == 0) {
                    failedtrial = 1;
                    break;
                }
            }
            if (failedtrial) {
                continue;
            }

            // If p mod e = 1 then gcd(p, e) > 1, so try again
            bignum_mod_short(p, e, &u);
```

```
            if (bignum_cmp_short(&u, 1) == EQUAL) {
                continue;
            }

            // Do expensive primality test
            if (rsa_check_prime(p) == PRIME ) {
                done = 1;
                break;
            }

        }
    }


    printf("\n");
    return (done ? count : -1);
}
```

4.1 Calcolo della chiave di Decrittazione

Pero ottenere d, ovvero la prima variabile della chiave privata, è necessario utilizzare l'estensione dell'algoritmo di Euclide, visto in precedenza per il calcolo del massimo comun divisore. La computazione dell'algoritmo richiede due parametri in ingresso, ovvero $\varphi(n)$ ed e, e porta ad un'unica variabile di uscita, denominata *inverso moltiplicativo*.

Il calcolo di $gcd(\varphi(n), e)$ viene effettuato calcolando una serie di zeri $x_0, x_1, \ldots$, dove

$$\begin{aligned} x_0 &= \varphi(n) \\ x_1 &= e \\ x_i &= x_{i-2} \bmod x_{i-1} \end{aligned}$$

La condizione che termina il ciclo iterativo è il ritrovamento di un $x_k = 0$. Il massimo comun divisore sarà quindi $gcd(\varphi(n), e) = x_{k-1}$.

L'estensione dell'algoritmo prevede poi, che per ogni x_i vengano calcolati i coefficienti a_i e b_i tali che

$$x_i = a_i \cdot x_0 + b_i \cdot x_1$$

Se $x_{k-1} = 1$, allora b_{k-1} è l'inverso moltiplicativo di $x_1 \pmod{x_0}$.

L'algoritmo implementato nel crittosistema, però, segue la procedura descritta da D. Knuth nell'*algoritmo X*, più efficiente a livello computazionale.

Dati due interi non negativi $\vec{u}$ e $\vec{v}$, questo algoritmo determina un vettore $(\vec{u_1}, \vec{u_2}, \vec{u_3})$ tale che $\vec{u} \cdot \vec{u_1} + \vec{v} \cdot \vec{u_2} = \vec{u_3} = gcd(\vec{u}, \vec{v})$. La computazione richiede l'ausilio di due vettori

$(\vec{v}_1, \vec{v}_2, \vec{v}_3)$ e $\left(\vec{t}_1, \vec{t}_2, \vec{t}_3\right)$, che saranno manipolati in modo tale che le equazioni

$$\vec{u} \cdot \vec{t}_1 + \vec{v} \cdot \vec{t}_2 = \vec{t}_3$$
$$\vec{u} \cdot \vec{u}_1 + \vec{v} \cdot \vec{u}_2 = \vec{u}_3$$
$$\vec{u} \cdot \vec{v}_1 + \vec{v} \cdot \vec{v}_2 = \vec{v}_3$$

rimangano vere per tutto il calcolo dell'algoritmo.

L'algoritmo, in realtà, ignora i vettori $\vec{u}_2$, $\vec{v}_2$ e $\vec{t}_2$ e attua dei meccanismi per prevenire i numeri negativi. Nel caso in cui l'inverso moltiplicativo non venga trovato, la procedura genera un errore.

Algoritmo 15: inversoMoltiplicativo

```
   Input: u⃗, v⃗
   Output: inv⃗
1  begin
2      u⃗_1 ← 1
3      u⃗_3 ← u⃗
4      v⃗_1 ← 0
5      v⃗_3 ← v⃗
6      t⃗_1 ← 0
7      t⃗_3 ← 0
8      bIterations ← 1
9      while v⃗_3 ≠ 0 do
10         q⃗ ← u⃗_3 / v⃗_3
11         t⃗3 ← u⃗_3 mod v⃗_3
12         w⃗ ← q⃗ · v⃗_1
13         t⃗_1 ← u⃗_1 + w⃗
14         u⃗_1 ← v⃗_1
15         v⃗_1 ← t⃗_1
16         u⃗_3 ← v⃗_3
17         v⃗_3 ← t⃗_3
18         bIterations ← −bIterations
19     if bIterations < 0 then
20         inv⃗ ← v⃗ − u⃗_1
21     else
22         inv⃗ ← u⃗_1
23     if u⃗_3 ≠ 1 then
24         return "Errore."
25     return inv⃗
```

```
int rsa_mod_inv(struct bn* inv, struct bn* u, struct bn* v) {
    int bIterations;
    int result;

    struct bn u1, u3, v1, v3, t1, t3, q, w;
    bignum_init(&u1);
    bignum_init(&u3);
    bignum_init(&v1);
    bignum_init(&v3);
    bignum_init(&t1);
    bignum_init(&t3);
    bignum_init(&q);
    bignum_init(&w);


    /*
     * Step X1: Initialise
     */
    u1.num[0] = 0x01;
    bignum_assign(&u3, u);
    bignum_assign(&v3, v);

    bIterations = 1;    /* Remember odd/even iterations */


    /*
     * Step X2: Loop
     */
    while (!bignum_is_zero(&v3)) {
        /*
         * Step X3: Divide & Subtract
         */
        bignum_udiv(&u3, &v3, &q, &t3);
        bignum_umul(&q, &v1, &w);
        bignum_add(&u1, &w, &t1);


        // Swap u1 = v1; v1 = t1; u3 = v3; v3 = t3
        bignum_assign(&u1, &v1);
        bignum_assign(&v1, &t1);
        bignum_assign(&u3, &v3);
        bignum_assign(&v3, &t3);
```

```
        bIterations = -bIterations;
    }

    if (bIterations < 0)
        bignum_sub(v, &u1, inv);
    else
        bignum_assign(inv, &u1);

    // Make sure u3 = gcd(u,v) == 1
    if (bignum_cmp_short(&u3, 1) != EQUAL) {
        result = 1;
        bignum_init(inv);
    } else
        result = 0;


    return result;
}
```

4.2 Procedura di crittazione e/o decrittazione

La procedura necessaria a decrittare un messaggio è identica a quella per criptarlo. Dal momento che l'equazione che restituisce il messaggio crittato $C \equiv M^e \pmod{n}$ è simile a quella per la decrittazione $D(C) \equiv C^d \pmod{n}$, è sufficente un unico algoritmo per effettuare entrambe le operazioni.

Il calcolo di $C \equiv M^e \pmod{n}$ necessita al più di $2\log_2 e$ moltiplicazioni e di $2\log_2 e$ divisioni utilizzando la seguente procedura, denominata *esponenziazione per moltiplicazioni e quadrati ripetuti*.

Algoritmo 16: powMod

```
   Input: b⃗, e⃗, n⃗
   Output: c⃗
 1 begin
 2     base⃗ ← b⃗ mod n⃗
 3     if base⃗ = 0 then
 4         return c⃗ ← 0
 5     exp⃗ ← e⃗
 6     c⃗ ← 1
 7     while exp⃗ ≠ 0 do
 8         if exp⃗ mod 2 ≠ 0 then
 9             c⃗ ← (c⃗ · base⃗) mod n⃗
10         base⃗ ← (base⃗ · base⃗) mod n⃗
11         Spostare exp⃗ a destra di 1bit
12     return c⃗
```

```
void rsa_pow_mod(struct bn* b, struct bn* e, struct bn* n,
    struct bn* c) {
    struct bn base, exp, tmp;
    int i, iteration;
    bignum_init(&base);
    bignum_init(&exp);
    bignum_init(&tmp);

    bignum_mod(b, n, &base);
    if ( bignum_is_zero(&base) ) {
        bignum_init(c);
        return;
    }
```

```
    bignum_assign(&exp, e);
    bignum_from_word(c, 0, 1);

    iteration = rsa_sizeof(&exp);
    for (i = 0; i < iteration; i++) {
        if ( (exp.num[i/WORD_SIZE] >> (i%WORD_SIZE)) & 0x01 ) {
            bignum_mul(c, &base, &tmp);
            bignum_mod(&tmp, n, c);
        }

        bignum_mul(&base, &base, &tmp);
        bignum_mod(&tmp, n, &base);
    }
}
```

La procedura implementata è stata ulteriormente migliorata dal punto di vista computazionale eliminando il ciclo *while* e sostituendolo con un ciclo *for*. La condizione di termine del ciclo è il raggiungimento, da parte della variabile contatore, della lunghezza effettiva[2] del vettore $\vec{exp}$. Così facendo, si evita di controllare, ad ogni ciclo iterativo, la non nullità del vettore $\vec{exp}$.

[2]Con lunghezza effettiva si intende il numero di bit meno significativi dopo il primo bit più significativo diverso da 0.

5 Analisi prestazionale

Per effettuare un'analisi prestazionale del crittosistema sviluppato, è doveroso tenere conto di diverse variabili. Tra queste spiccano il tempo e lo spazio di memoria necessari per la generazione delle chiavi.

5.1 Generazione di numeri primi

Il processo che più di tutti appesantisce la generazione delle chiavi è sicuramente quello che porta alla generazione di un numero primo. Infatti, come approfondito nel capitolo "Test di primalità", viene trovato, in media, un numero primo lungo 1*kbit* ogni 355 numeri interi dispari casuali. Questo vuol dire che dovranno essere effettuati altrettanti test più o meno dispendiosi, per determinare la primalità di un numero dispari.

Le lunghezze interessate per questo tipo di test sono cinque, ovvero:

- 128 bit;
- 256 bit;
- 512 bit;
- 1024 bit;
- 1536 bit.

Le prime tre rappresentano quello che è stato lo standard richiesto negli anni precedenti, la quarta rappresenta lo standard odierno, mentre l'ultima rappresenta lo standard richiesto nel prossimo decennio. Si ricorda che le lunghezze imposte ai numeri primi sono la metà rispetto alla lunghezza della chiave n.

I test sono stati effettuati in ambiente Linux, più in particolare su un server Ubuntu equipaggiato di:

- n.4 CPU AMD EPYC™ 7501, 32 core, 64 threads, con una frequenza base di 2.0GHz;
- 8GB di RAM di tipo DDR4.

Di seguito viene riportata la tabella contenente i dati registrati durante i test. Sono stati generati, per ogni lunghezza, 100 numeri primi casuali e per ognuno di questi sono stati registrati:

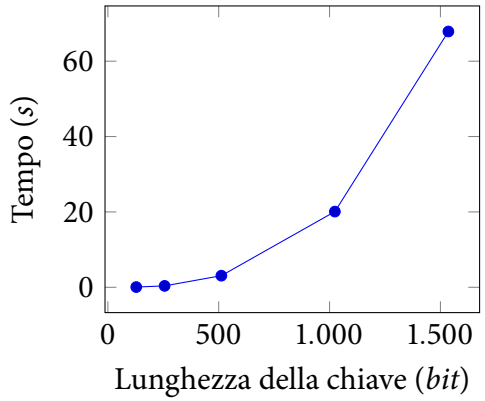
- il tempo medio necessario alla generazione di un numero primo;
- il numero medio di interi generati prima di trovare un numero primo;
- il tempo medio necessario a valutare la primalità di ogni singolo numero intero generato.

È interessante aggiungere una quinta colonna alla tabella il tempo impiegato dal calcolatore ad eseguire i test di primalità per ciascun bit. In questo modo si verifica come la lunghezza della chiave incida sulla complessità di calcolo dell'algoritmo.

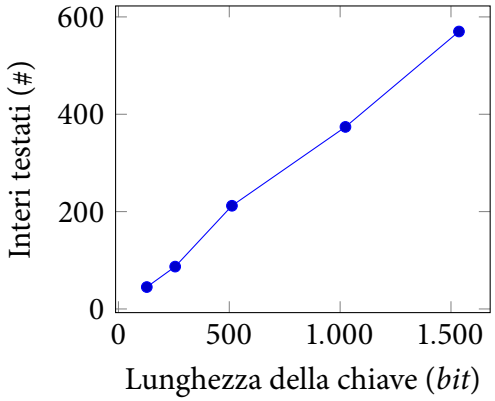
Lunghezza (bit)	Tempo (s)	# di interi testati (#)	Tempo per numero ($ms/\#$)	Tempo per bit ($ms/\# \cdot bit$)
128	0,063	45	1,7	0,0133
256	0,374	87	4,9	0,0191
512	3,082	212	16,4	0,0320
1024	20,087	374	58,1	0,0567
1536	67,877	570	122,8	0,0799

Tabella 5.1: Misurazione delle prestazioni per la generazione di numeri primi di lunghezza indicata.

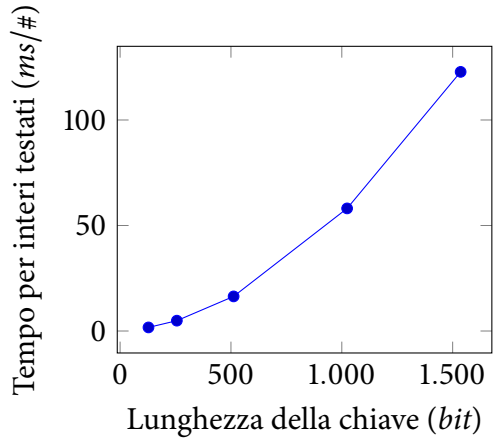
Come si evince dai grafici rappresentati nella figura 5.1, il tempo medio necessario a generare un numero primo cresce, con l'aumentare della lunghezza del numero stesso, seguendo un andamento esponenziale. Anche il tempo medio impiegato per testare ogni singolo numero generato prima di trovare un numero primo (fig. 5.1c) segue un andamento esponenziale, ma la forza con cui la curva sale è minore rispetto al grafico precedente.



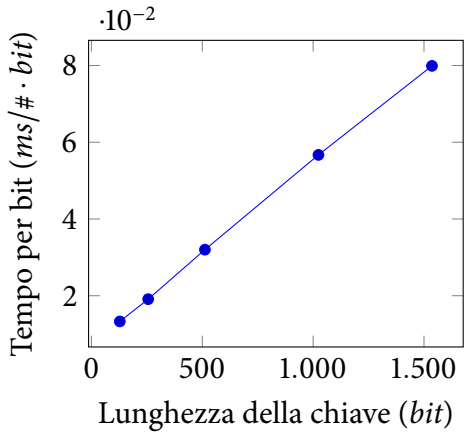
(a) Corrispondenza lunghezza / tempo



(b) Corrispondenza lunghezza / interi testati



(c) Corrispondenza lunghezza / tempo per interi testati



(d) Corrispondenza lunghezza / tempo per bit

Figura 5.1: Prestazione test di primalità.

La curva che indica il numero di interi generati e testati prima di trovare un numero primo (fig. 5.1b) e quella che indica il tempo medio necessario ad effettuare il test per ciascun bit generato (fig. 5.1d) seguono, tuttavia, un andamento che può essere definito lineare.

5.2 Generazione della coppia di chiavi

La creazione dei numeri primi richiede la quasi totalità del tempo di esecuzione del processo che genera la coppia di chiavi. Nella tabella seguente, si può notare tale occupazione in termini percentuali. Nonostante una lieve retrocessione, la percentuale di tempo dedicato al calcolo dei numeri primi p e q tende a prevalere sul resto, fino a rendere quasi irrilevante il tempo necessario ad effettuare tutti i controlli a generare il resto della coppia di chiavi.

Lunghezza (*bit*)	Tempo intero processo (*s*)	Tempo calcolo primi (*s*)	Tempo rimanente (*s*)	Occupazione calcolo primi (%)
256	0,1402	0,126	0,0142	90%
512	0,8360	0,748	0,0880	89%
1024	7,0768	6,164	0,9128	87%
2048	40,8828	40,174	0,7088	98%
3072	136,9689	135,754	1,2149	99%

Tabella 5.2: Misurazione dei tempi medi di generazione delle coppie di chiavi di lunghezza indicata.

5.3 Spazio di memoria

Mantenendo la stessa configurazione del server Ubutnu illustrata in precedenza, si è studiata l'occupazione di memoria dovuta all'implementazione del crittosistema. Il file eseguibile, compilato con il compilatore *GNU Compiler Collection*, occupa $32KB$ della memoria di massa. L'occupazione della memoria RAM, quantificata mentre il processo è in esecuzione, è di circa $4700B$. Per avere una stima più accurata, sarebbe necessario compilare la mappa di memoria.

6
Conclusione

Tra la prime difficoltà incontrate nello sviluppo del crittosistema, spicca l'implementazione degli algoritmi di operazioni matematiche tra numeri che, nei capitoli precedenti, abbiamo visto essere composti da *array di interi*. Ciò vuol dire che le operazioni canoniche a disposizione non erano utilizzabili senza adattamenti perfezionati *ad hoc*. I fattori che hanno condizionato la scelta degli algoritmi utilizzati non sono stati esclusivamente di natura prestazionale, ma un giusto compromesso tra complessità computazionale e utilizzo dello spazio a disposizione. Non bisogna dimenticare che questo specifico crittosistema è pensato e sviluppato per un funzionamento su *sistemi embedded*, quindi in condizioni di memoria RAM molto limitata.

Una volta definiti gli algoritmi che permettessero l'interazione matematica tra le variabili, e dopo aver cercato di ridurre al minimo le operazioni e le variabili temporanee, si è passati a definire tutti quegli algoritmi che servono al funzionamento, in senso stretto, del crittosistema, quindi algoritmi per la generazione di numeri casuali, per il controllo di primalità, per la generazione finale delle chiavi pubbliche e private. Anche per tali algoritmi si è cercato un compromesso che rendesse accettabile il tempo di attesa, senza inficiare troppo sullo spazio di memoria.

Tra le tante difficoltà, il raggiungimento dei vari obiettivi prefissati è stato di aiuto per risolvere gli ulteriori errori e *bug* che, bene o male, sono spesso presenti nel cammino che porta da un'idea allo sviluppo del prodotto. Uno scoglio molto importante è stato lo sviluppo dell'algoritmo adibito al test di primalità dei numeri casuali: da una complessità computazionale che oscillava tra il polinomiale e l'esponenziale, si è riusciti, attraverso a continue modifiche nell'algoritmo, a portare l'esecuzione a tempi di attesa accettabili.

Senza ombra di dubbio sono diverse le modifiche ed i miglioramenti che possono essere attuati al crittosistema sviluppato, ad esempio alcuni algoritmi possono essere riscritti in modo tale da essere ancor più performanti, oppure è possibile convertire in iterativi i pochi algoritmi ricorsivi presenti, preferibili in quanto creano meno variabili con un conseguente risparmio di memoria.

Complessivamente, non è stato un lavoro banale, ma nemmeno troppo difficile. La comprensione degli argomenti trattati, il capire i meccanismi, nascosti agli utenti finali, che permettono il funzionamento del sistema ha fatto sì che si instaurasse una curiosità molto profonda del sistema stesso. Motivo per cui il lavoro può considerarsi concluso se fine a se stesso, ma è da pensare come l'inizio di un nuovo cammino, se si pensa a tutte le applicazioni che fanno da corollario e che completano il crittosistema RSA.

A

Metodi di attacco al Crittosistema RSA

Diverse sono le modalità di attacco al crittosistema, a partire dalla fattorizzazione di n, ovvero il metodo più ovvio, fino ad arrivare ad attacchi più sofisticati.

A.1 Fattorizzazione

Dal momento che la variabile n è data dal prodotto di due numeri primi, la prima possibilità consiste nel dividerla con ogni intero dispari $p \leq \lfloor \sqrt{n} \rfloor$. Questo metodo diventa irragionevole nel momento in cui la variabile n supera la soglia di 10^{12}. In questo momento storico, dove è suggerito utilizzare delle chiavi di lunghezza $2kbit$, tale soglia è stata superata all'incirca di 10^{605} volte.

A.1.1 Algoritmo $p-1$ di Pollard

Siano n, B due numeri interi. Supponiamo che esista un numero primo p, divisore di n, tale che se $p-1 = \prod_{i=1}^{k} q_i^{\alpha_i}$, allora $q_i^{\alpha_i} \leq B$. Per cui $p-1$ è divisore di $B!$. Esiste, allora, un numero $j_0 \in [2, B!]$ tale che $j_0 = k(p-1)$.

Per il piccolo Teorema di Fermat, vale che $2^{j_0} \equiv 1 \pmod{p}$. Quindi p risulta essere divisore di $gcd(2^{j_0} - 1, n)$.

Se $gcd(2^{j_0} - 1, n) < n$, allora $gcd(2^{j_0} - 1, n)$ è un fattore di n.

Algoritmo 17: $p - 1$ di Pollard

Input: n, B
Output: d in caso di successo, -1 altrimenti

```
1 begin
2     a ← 2
3     for j ← 2 to B do
4         a ← 2^j − 1
5         d ← gcd(a, n)
6     if 1 < d < n then
7         return d
8     return −1
```

Analizzando l'algoritmo, si evince che sia di tipo *Las Vegas*, ovvero il cui output è sempre corretto, ma il tempo di esecuzione è probabilistico. Perché l'algoritmo funzioni, n deve ammettere un divisore primo p, tale che $p - 1$ sia costituito da potenze di primi piccoli, in particolare minori o uguali a B.

Un modo abbastanza immediato per rendere il crittosistema RSA immune da questo tipo di attacco consiste nel generare due numeri primi p e q tali che $p = 2p_1+1$ e $q = 2q_1+1$, con p_1 e q_1 primi grandi.

A.1.2 Algoritmo ρ di Pollard

L'algoritmo ρ di Pollard è un metodo probabilistico di fattorizzazione basato sulla seguente idea.

Sia p il divisore primo più piccolo di n. Se esistono $x, x' \in \mathbb{Z}_n$ diversi tra loro e tali che $x \equiv x' \pmod{p}$, allora

$$p \leq gcd(x - x', n) \leq n$$

È importante notare che, presi casualmente x e x', è possibile calcolare $gcd(x - x', n)$ senza conoscere p. Inoltre, se $x, x' \in \mathbb{Z}_p$, $1 \leq p \leq gcd(x - x', n)$.

Per fare ciò, si sceglie $X \subseteq \mathbb{Z}_n$ e, per ogni coppia (x, x') di elementi distinti di X, si calcola $gcd(x - x', n)$. L'algoritmo ha successo se $x \mapsto x \bmod p$ ammette una collisione in X.

Utilizzando il *paradosso del compleanno*, ciò si verifica con una probabilità del 50% se $|X| \simeq 1.17\sqrt{p}$. Tuttavia, siccome p è sconosciuto, la collisione è determinata solo valutando $gcd(x - x', n)$ per ogni coppia (x, x') di elementi distinti di X.

Pertanto, per determinare una collisione, il numero dei *massimi comun divisori* da calcolare è

$$\binom{|X|}{2} > \frac{p}{2}$$

che è un numero elevato. Per ridurre il numero di questi calcoli, che la memoria da utilizzare, l'algoritmo ρ di Pollard procede come segue:

1. Si considera $f \in \mathbb{Z}[x]$ e $x_1 \in \mathbb{Z}_n$; generalmente, $f(x) = x^2 + 1$;
2. Si consideri la successione definita da $x_{j+1} \equiv f(x_j) \pmod{n}$ per $2 \leq j \leq m - 1$;
3. Ogni volta che si determina un x_j, si valuta $gcd(x_j - x_i, n)$ per ogni $j < i$.

Si può dimostrare che

$$x_i \equiv x_j \pmod{p} \implies \forall \delta \in \mathbb{N} : x_{i+\delta} \equiv x_{j+\delta} \pmod{p}$$

Ci viene, quindi, in aiuto un trucco, detto "di Floyd", che afferma: se $x_i \equiv x_j \pmod{p}$, allora esiste $i' \in i, i+1, \ldots, j-1$ tale che $x_{i'} \equiv x_{2i'} \pmod{p}$.

Algoritmo 18: ρ di Pollard

```
Input: n, x_1
Output: p in caso di successo, −1 altrimenti
1  begin
2      x ← x_1
3      x' ← f(x) mod n
4      p ← gcd(x − x', n)
5      while p = 1 do
6          x ← f(x) mod n
7          x' ← f(f(x')) mod n
8          p ← gcd(x − x', n)
9      if p ≠ n then
10         return p
11     return −1
```

Nell'algoritmo si cercano collisioni del tipo $x_{i'} \equiv x_{2i'} \pmod{p}$ per qualche $i' \in \{i, i+1, \ldots, j-1\}$. Quindi, all'iterazione i-esima si calcola solo $gcd(x_{2i} x_i, n)$. Il numero di iterazioni per determinare un fattore primo di n è di circa $\sqrt{p}$.

La probabilità di successo dell'algoritmo è circa $\frac{p}{n} < \frac{1}{\sqrt{n}}$, mentre la sua complessità è pari a $O(\sqrt[4]{n} \cdot \ln n)$.

A.1.3 Algoritmo di Dixon

Nel 1981, John D. Dixon, professore di matematica all'Università di Carleton, pubblica un algoritmo per la fattorizzazione di numeri interi. L'idea di base è molto semplice:

determinare due interi x e y tali che

$$\begin{cases} x \not\equiv \pm y \pmod{n} \\ x^2 \equiv y^2 \pmod{n} \end{cases}$$

Pertanto, n è divisore di $(x+y)(x-y)$, ma non divide $x+y$ o $x-y$. Allora $gcd(n, x-y)$ e $gcd(n, x+y)$ sono fattori non banali di n.

L'algoritmo opera nel modo seguente:

1. Sia $\mathcal{B} = \{p_1, \ldots, p_b\}$ un sottoinsieme di numeri primi;

2. Siano $z_1, \ldots, z_c$ degli interi casuali, con $c > b$. Per ogni z_i, si calcoli $z_i^2 \bmod n$. Generalmente, gli interi scrivibili come $j + \lceil \sqrt{kn} \rceil$ con $j = 0, 1, \ldots$ e $k = 1, 2, \ldots$, tendono a produrre i corrispondenti $z_i^2 \bmod n$ relativamente piccoli, quindi completamente fattorizzabili rispetto ai primi di $\mathcal{B}$.

3. Per ogni $j \in [1, c]$,
$$z_j^2 \equiv p_1^{\alpha_{1j}} \cdot p_2^{\alpha_{2j}} \cdots p_b^{\alpha_{bj}} \pmod{n}$$
si considera il vettore di $\mathbb{Z}_2^b$ definito da
$$\vec{a}_i = (\alpha_{1j} \bmod 2, \alpha_{2j} \bmod 2, \ldots, \alpha_{bj} \bmod 2)$$

4. Poiché $c > b$, i vettori $\vec{a}_1, \ldots, \vec{c}_1$ sono linearmente dipendenti. Quindi esiste $X \subseteq \{1, \ldots, c\}$ tale che $\sum_{j \in X} \vec{a}_j = \vec{0}$; pertanto
$$\prod_{j \in X} z_j^2 \equiv \prod_{i=1}^{b} p_i^{\sum_{j \in X} \alpha_{ij}} \pmod{n}$$
Siccome $\sum_{j \in X} \vec{a}_j = \vec{0}$, allora
$$\sum_{j \in X} a_{ij} = 2k_i, \ \forall j \in X$$
Posto $x = \prod_{j \in X} z_j$ e $y = \prod_{i=1}^{b} p_i^{k_i}$ vale che
$$x^2 \equiv y^2 \pmod{n}$$

5. Dal risultato del punto precedente, si procede al calcolo di $gcd(n, x-y)$ o di $gcd(n, x+y)$.

A.2 Altri metodi di attacco

A.2.1 Fattorizzazione attraverso $\varphi(n)$

Forzare il crittosistema RSA equivale ad essere in grado di calcolare la funzione *toziente* $\varphi(n)$. Se, oltre alla chiave n, si è a conoscenza della funzione toziente, allora è facilmente computabile la fattorizzazione di n.

Sia $n = p \cdot q$, con p e q primi distinti tra loro. Allora

$$\begin{aligned}\varphi(n) &= (p-1)\cdot(q-1)\\ &= p\cdot q - p - q + 1\\ &= n - (p+q) + 1\end{aligned}$$

Invertendo di posto $\varphi(n)$ e $p+q$ si ottiene

$$p + q = n - \varphi(n) + 1$$

Quindi sostituiamo q con $\frac{n}{p}$ ed otteniamo

$$\begin{aligned}p + \frac{n}{p} &= n - \varphi(n) + 1\\ p^2 + n &= (n - \varphi(n) + 1)\, p\\ p^2 - (n - \varphi(n) + 1)\, p + n &= 0\end{aligned}$$

Detto $k = (n - \varphi(n) + 1)$, si ottengono le soluzioni:

$$\begin{aligned}p &= \frac{k \pm \sqrt{k^2 + 4n}}{2}\\ q &= \frac{2n}{k \pm \sqrt{k^2 + 4n}}\end{aligned}$$

La complessità computazionale richiesta per calcolare p e q a partire dalla conoscenza di n e di $\varphi(n)$ è pari a $O(\log^3 n)$.

A.2.2 Esponente di decifratura

Come la conoscenza del valore della funzione toziente porta alla fattorizzazione di n, anche la conoscenza dell'esponente di decifratura d porta allo stesso risultato, ma in tempi polinomiali.

Sia $n = p \cdot q$, con p e q primi e distinti tra loro, e siano e e d gli esponenti rispettivamente di cifratura e di decifratura di un generico utente che utilizza il crittosistema RSA. Allora

$$ed - 1 = 2^s \cdot r \equiv 0 \pmod{\varphi(n)}$$

con r dispari. Se w è un intero tale che $gcd(w, n) = 1$, dal Teorema di Eulero discende che

$$w^{2^s \cdot r} \equiv 1 \pmod{n}$$

Sia $t = min\{0, \ldots, s\}$ tale che $w^{2^t \cdot r} \equiv 1 \pmod{n}$. Se $w^{2^{t-1} \cdot r} \not\equiv -1 \pmod{n}$ per $t > 0$, allora $w^{2^{t-1} \cdot r}$ è una radice quadrata non banale di 1 in $\mathbb{Z}_n$. Pertanto

$$1 < gcd(w^{2^{t-1} \cdot r}, n) < n$$

cioè $gcd(w^{2^{t-1} \cdot r}, n) \in [p, q]$ e quindi n è fattorizzato.

Se, però, w è un intero tale che $gcd(w, n) = 1$ e vale una delle seguenti congurenze

1. $w^r \equiv 1 \pmod{n}$
2. $w^{2^t \cdot r} \equiv -1 \pmod{n}$ per $t \in [0, s-1]$

il procedimento illustrato sopra non è più valido. Questo perché w è una base rispetto alla quale n è uno pseudoprimo di Eulero.

La probabilità di fattorizzare n corrisponde alla probabilità di trovare una base rispetto alla quale n non è uno pseudoprimo di Eulero. Tale probabilità è maggiore o uguale ad $1/2$.

A.2.3 Attacco di Wiener

L'attacco di Wiener prende il nome dal suo inventore Norbert Wiener (1894-1964), matematico e statistico statunitense. Tale attacco al crittosistema si basa su una debolezza del sistema stesso, in particolare sulla scarsa lunghezza dell'esponente di decifratura. Infatti, minore è la lunghezza della chiave d, minore è il tempo necessario per decrittare il messaggio cifrato. Questo perché, per un modulo di lunghezza fissata, il tempo di decifrazione è proporzionale alla lunghezza in bit dell'esponente scelto.

L'attacco si basa fondamentalmente su tre concetti: il teorema di Wiener, il teorema di Legendre e l'algoritmo delle frazioni continue.

Teorema A.1 (Teorema sulla convergenza). Siano a, b, c e d numeri interi tali che $gcd(a, b) = gcd(c, d) = 1$. Se

$$\left| \frac{a}{b} - \frac{c}{d} \right| < \frac{1}{2d^2}$$

allora $\frac{c}{d}$ è uno dei convergenti di $\frac{a}{b}$.

Teorema A.2 (Teorema di Wiener). Sia $n = p \cdot q$, con p e q primi tali che $q < p < 2q$. Siano $d \geq 1$ ed $e < \varphi(n)$ tali che $ed \equiv 1 \pmod{\varphi(n)}$.

Se $3d < \sqrt[4]{n}$ allora $\frac{t}{d}$ è uno dei convergenti dello sviluppo mediante frazioni continue di $\frac{e}{n}$.

Dimostrazione. Poiché $ed \equiv 1 \pmod{\varphi(n)}$, segue che

$$\exists t \in \mathbb{Z} \mid ed - 1 = t\varphi(n)$$

Questo implica che $t\varphi(n) < ed < \varphi(n)d$ e $t < d$, che a loro volta implica che

$$3t < 3d < \sqrt[4]{n} \tag{A.1}$$

Dato che $n = pq$ e che $pq > q^2$, si ha che $q < \sqrt{n}$; inoltre da $p < 2q$ segue che

$$0 < n - \varphi(n) = p + q - 1 < 2q + q - 1 < 3q < 3\sqrt{n}$$

Dal risultato (A.1) segue che

$$0 < d\,(n - \varphi(n)) < 3k\sqrt{n} < \sqrt[4]{n^3}$$

Dal teorema sulla convergenza, vediamo che

$$\begin{aligned}\left|\frac{e}{n} - \frac{t}{d}\right| &= \left|\frac{ed - tn}{dn}\right| = \left|\frac{ed - 1 + 1 - tn}{dn}\right| = \left|\frac{t\varphi(n) + 1 - kn}{dn}\right| \\ &= \left|\frac{1 - t\,(n - \varphi(n))}{dn}\right| \leq \frac{t\,(n - \varphi(n))}{dn} \leq \frac{\sqrt[4]{n^3}}{dn} \\ &= \frac{1}{d\sqrt[4]{n}} \leq \frac{1}{3d^2} < \frac{1}{2d^2}\end{aligned}$$

Pertanto, $\frac{t}{d}$ è uno dei convergenti dell'espansione mediante frazioni continue di $\frac{e}{n}$. □

L'algoritmo che segue è lo pseudocodice per un attacco di Wiener ad un crittosistema RSA. Il richiamo alla funzione *algoritmo di Euclide esteso* si riferisce all'estensione dell'algoritmo di Euclide per il calcolo del massimo comun divisore, descritta nel paragrafo 4.1

relativo al *calcolo della chiave di decrittazione*, a pagina 38.

Algoritmo 19: wienerAttack

Input: n, e
Output: (p, q) in caso di successo, -1 altrimenti

```
1  begin
2      (q_1, ..., q_m; r_m) ← algoritmo di Euclide esteso(n, b)
3      c_0 ← 1
4      c_1 ← q_1
5      d_0 ← 0
6      d_1 ← 1
7      for j ← 1 to m do
8          n' ← (d_j e − 1)/c_j
           /* Se c_j/d_j è il convergente corretto, allora n' = φ(n) */
9          if n' ∈ ℤ then
10             (p, q) ← le soluzioni dell'equazione x^2 − (n − n' + 1)x + n = 0
11             if p, q ∈ ℤ^+ e p < n e q < n then
12                 return (p, q)
13         j ← j + 1
14         c_j ← q_j c_{j−1} + c_{j−2}
15         d_j ← q_j d_{j−1} + d_{j−2}
16     return −1
```

Bibliografia

[1] Elaine Baker e Quynh Dang. «Recommendation for Key Managment. Part 3: Application Specific Key. Management Guidance». In: (2015). URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57Pt3r1.pdf.

[2] Stefano Diomedi. «Attacco di Wiener ad RSA mediante frazioni continue». online. 2013-2014. URL: https://amslaurea.unibo.it/8708/1/Diomedi_Stefano_tesi.pdf.

[3] Donald Ervin Knuth. *The Art of Computer Programming. Seminumerical Algorithms*. 3ª ed. Vol. 2. Addison-Wesley, 1997. ISBN: 0-201-89684-2.

[4] Alessandro Montinaro e Pierluigi Rizzo. «Introduzione alla Crittografia». In: (2019). URL: http://siba-ese.unisalento.it/index.php/quadmat/article/download/21251/17955.

[5] Ronald Linn Rivest, Adi Shamir e Leonard Adlemann. «A Method for Obtaining Digital Signatures and Public-Key Cryptosystems». In: (1977). URL: https://people.csail.mit.edu/rivest/Rsapaper.pdf.

[6] *RSA (cryptosystem)*. URL: https://en.wikipedia.org/wiki/RSA_(cryptosystem). (ultimo accesso: 10 settembre 2020).

[7] *RSA Algorithm*. URL: https://www.di-mgt.com.au/rsa_alg.html. (ultimo accesso: 10 settembre 2020).

Ringraziamenti

A conclusione di questo lavoro di tesi, è doveroso porre i miei più sentiti ringraziamenti alle persone che ho avuto modo di conoscere in questo periodo della mia vita e che mi hanno aiutato a crescere sia dal punto di vista professionale che umano.

Ringrazio anzitutto il mio relatore, il professor Zingirian, per avermi guidato nella stesura di questo lavoro, per avermi trasmesso la passione e l'entusiasmo necessari affinché la tesi prendesse forma giorno dopo giorno.

Non so se trovo le parole giuste per ringraziare i miei genitori e la mia famiglia, però vorrei che questo mio traguardo raggiunto, per quanto possibile, fosse un premio anche per loro e per i sacrifici che hanno fatto. Un infinito grazie per esserci sempre, per sostenermi, per avermi insegnato ciò che è "giusto" e ciò che non lo è. Grazie ai loro consigli e alle loro critiche che mi hanno fatto crescere.

"Ti preoccupi troppo per ciò che era e ciò che sarà.
C'è un detto: ieri è storia, domani è un mistero, ma oggi...
è un dono; per questo si chiama Presente."

Questa tesi è stata scritta usando LaTeX, originariamente sviluppato da Leslie Lamport e basato sul TeXdi Donald Knuth. Il testo del corpo è riportato in 12 punti MinionPro, basato sull'omonimo Minion disegnato da Robert Slimbach e ispirato allo stile tardo-rinascimentale; il carattere è stato progettato per il corpo del testo in uno stile classico, leggermente condensato e con grandi aperture per aumentare la leggibilità.