Sifat Prima Terhadap Fungsionalitas Algoritma RSA

Kamal Mahmudi

Mahasiswa Jurusan Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
Labtek V, Jalan Ganeca 10 Bandung
Email: if17111@students.if.itb.ac.id

ABSTRAK

Algoritma RSA merupakan salah satu algoritma enkripsi yang menggunakan kunci publik dan kunci privat. Algoritma ini menurunkan Teorema Fermat dan Fungsi Euler yang bermain-main pada sifat prima suatu bilangan dan sifat relatif prima bilangan tersebut terhadap bilangan yang lain. Secara umum algoritma RSA dimulai dengan membangkitkan dua buah bilangan prima (p dan q) yang berbeda yang kemudian akan dihitung nilai dari fungsi euler dari hasil perkalian kedua bilangan tersebut (n), baru kemudian dihasilkan kunci publik (e) dan kunci privat (d) yang akan digunakan dalam proses enkripsi dekripsi.

Mengingat bahwa hasil perkalian kedua bilangan prima tersebut harus disertakan pada kunci publik, semakin besar bilangan prima semakin susah hasil perkalian tersebut untuk difaktorkan kembali menjadi kedua bilangan tersebut. Membangkitkan bilangan prima besar menghabiskan banyak waktu dan ruang memori pada *hardware*. Sementara pengujian bilangan prima tidak menjamin suatu bilangan pasti merupakan bilangan prima atau tidak.

Makalah ini akan membahas pentingnya sifat bilangan prima dan relatif prima dan melihat kelebihan dan kekurangan dari berbagai solusi permasalahan bilang prima besar.

Kata kunci: Bilangan Prima, Fungsi Euler, Pembangkit Bilangan Prima, Teorema Fermat, RSA.

1. PENDAHULUAN

Penggunaan algoritma RSA yang menurunkan Teorema Fermat dan Fungsi Euler bermain-main pada sifat prima suatu bilangan. Dari hal tersebut dapat disimpulkan bahwa proses pembangkitan kunci sangat bergantung pada proses pembangkitan dua buah bilangan prima pada awal proses. Mengingat bahwa hasil perkalian kedua bilangan prima tersebut harus disertakan pada kunci publik, semakin besar bilangan prima semakin susah hasil perkalian tersebut untuk difaktorkan kembali menjadi kedua bilangan tersebut. Para pengguna algoritma RSA dapat saja menggunakan daftar bilangan prima sebagai masukan awal proses pembangkitan kunci, namun tentu saja penggunaan daftar bilangan prima memiliki beberapa kelemahan seperti sifatnya yang sangat bergantung pada panjang bilangan prima yang dimiliki daftar atau bahkan tidak membuka kemungkinan munculnya bilangan di luar daftar. Sementara bilangan prima sendiri tidak memiliki fungsi yang memetakan suatu indeks terhadap nilai suku pada indeks tersebut sehingga tidak dapat dibangkitkan begitu saja.

Salah satu cara untuk mendapatkan bilangan prima besar yang akan digunakan pada pembangkitan kunci algoritma RSA adalah dengan mengambil acak suatu bilangan bulat besar dan kemudian dilakukan pengujian dengan menggunakan metode Lehman apakah bilangan tersebut memiliki kemungkinan sebagai bilangan prima dengan batas toleransi tertentu. Seperti dikatakan sebelumnya, metode ini tidak menjamin

bilangan tersebut adalah bilangan prima, namun hanya memberi kemungkinan yang relatif besar sesuai keinginan pengguna.

Sementara algoritma-algoritma yang dapat membangkitkan daftar bilangan prima besar memiliki kelebihan dan kekurangan yang berbeda-beda namun tetap bergantung pada keterbatasan ruang memori dan waktu yang dapat disediakan.

2. Algoritma RSA

Algoritma RSA merupakan algoritma kriptografi kunci-publik yang dibuat oleh 3 orang peneliti dari Massachussets Institute of Technology pada tahun 1976. Diambil dari nama pembuatnya Ron Rivest, Adi Shamir, dan Leonard Adleman, sebenarnya algoritma RSA memanfaatkan Fungsi Euler dan bentuk umum dari Teorema Fermat.

2. 1. Teorema Fermat

Secara umum teorema ini berbunyi: Jika p adalah bilangan prima dan m adalah bilangan bulat yang tidak habis dibagi dengan p, yaitu PBB(m, p) = 1, maka

$m^{p-1} \equiv 1 \pmod{p}$ ---------- (1)

2. 2. Fungsi Euler

Fungsi Euler ϕ mendefinisikan $\phi(n)$ untuk $n \geq 1$ menyatakan jumlah bilangan bulat positif yang lebih kecil dari n dan relatif prima terhadap n.

Dengan memperhatikan Teorema Fermat dan definisi dari Fungsi Euler, dapat diturunkan sebuah bentuk umum dari Teorema Fermat yaitu jika PBB(m, n) = 1, maka

$m^{\phi(n)} \equiv 1 \pmod{n}$ ---------- (2)

2. 3. Algoritma RSA

Seperti telah disinggung sebelumnya, algoritma ini diturunkan dari Fungsi Euler dan Teorema Fermat serta memanfaatkan sifat-sifat dari aritmatika modulo. Berikut adalah proses penurunan algoritma dimulai dari persamaan (2).

Berdasarkan sifat $a^k \equiv b^k \pmod{n}$ maka persamaan (2) dapat ditulis menjadi

$m^{k\phi(n)} \equiv 1^k \pmod{n}$ ---------- (3)

atau

$m^{k\phi(n)} \equiv 1 \pmod{n}$ ---------- (4)

Berdasarkan sifat $ac \equiv bc \pmod{n}$ jika $a \equiv b \pmod{n}$, maka perkalian persamaan (4) dengan m akan menghasilkan

$m^{k\phi(n)+1} \equiv m \pmod{n}$ ---------- (5)

Misalkan

$e \cdot d = k\phi(n) + 1$ ---------- (6)

Dengan mensubtitusikan persamaan (6) ke dalam persamaan (5) diperoleh

$m^{e \cdot d} \equiv m \pmod{n}$ ---------- (7)

yang artinya perpangkatan m dengan e diikuti dengan perpangkatan dengan d dan dilakukan operasi modulo terhadap n akan menghasilkan m semula. Sehingga enkripsi dan dekripsi dapat dirumuskan sebagai berikut

$E_e(m) = c \equiv m^e \pmod{n}$---------- (8)

$D_d(c) = m \equiv c^d \pmod{n}$---------- (9)

Karena $e \cdot d = d \cdot e$, maka enkripsi diikuti dekripsi ekivalen dengan dekripsi diikuti enkripsi

$D_d(E_e(m)) = E_e(D_d(m)) \equiv m^{e \cdot d} \pmod{n}$ ----- (10)

Oleh karena $m^{e \cdot d} \equiv (m + jn)^{e \cdot d} \pmod{n}$ untuk sembarang bilangan bulat j, maka transformasi yang dihasilkan dapat bersifat banyak ke satu. Agar transformasi dapat bersifat satu ke satu sehingga enkripsi dan dekripsi dapat berjalan sesuai dengan persamaan (8) dan (9) maka m harus berada pada himpunan bilangan bulat positif yang lebih kecil dari n.

2. 4. Pembangkitan Pasangan Kunci

Secara umum pasangan kunci algoritma RSA dapat dibangkitkan dengan cara berikut:

a. Menentukan dua buah bilangan prima p dan q.
b. Menghitung $n = p \cdot q$
c. Menghitung $\phi(n) = (p - 1)(q - 1)$
d. Menentukan e yang relatif prima terhadap $\phi(n)$
e. Menghitung d = ——— dengan mencoba nilai k satu-persatu.

3. PEMBANGKIT BILANGAN PRIMA

Ada berbagai metode yang dapat digunakan untuk menghasilkan sebuah bilangan prima. Untuk menghasilkan bilangan prima yang besar dengan

menggunakan ruang memori dan waktu. Secara umum pembangkitan bilangan prima dapat dikelompokkan menjadi dua, yaitu dengan membangkitkan bilangan prima dari bilangan prima terkecil dengan pengujian yang menghasilkan 100% bilangan prima atau dengan membangkitkan bilangan acak dan menguji kemungkinan bilangan tersebut prima.

3. 1. Alternatif Satu

Secara umum alternatif ini akan membangkitkan tabel bilangan prima sehingga untuk mengambil sebuah bilangan prima cukup diambil satu dari beberapa bilangan yang terdapat pada tabel. Pada alternatif ini, dapat digunakan beberapa metode yang memiliki kelebihan dan kekurangan tersendiri.

Pertama, digunakan teknik yang akan membagi sebuah bilangan yang akan diuji dengan semua bilangan bulat positif yang lebih kecil dari akar bilangan tersebut. Cara ini dapat disebut *brute force* karena mencoba setiap kemungkinan yang ada yang tentunya semakin besar nilai bilangan yang akan diuji maka semakin besar pula waktu yang dibutuhkan untuk menguji dikarenakan semakin banyak bilangan bulat yang akan digunakan sebagai pembagi. Namun cara ini dapat dibilang tidak memakan ruang memori karena hanya membagi dengan bilangan bulat positif yang lebih kecil dari akar bilangan tersebut.

Cara kedua tidak jauh dengan cara pertama namun digunakan pembagi yang jauh lebih sedikit. Cara ini hanya menggunakan bilangan prima yang telah dibuktikan sebelumnya sebagai pembagi untuk bilangan baru yang lebih besar yang akan diuji. Cara ini bergantung pada media penyimpanan bilangan prima yang pernah dihasilkan yang mungkin saja ditempatkan pada memori atau pada file eksternal. Dari sisi kandidat, kandidat yang akan diuji hanya angka 2 dan bilangan ganjil. Cara ini memangkas waktu lebih jauh jika menggunakan memori sebagai tempat penyimpanan.

Cara lain yang dapat dipertimbangkan adalah mengunakan pendekatan Teori Sieve yang membuat sebuah array sepanjang kandidat prima terbesar ditambah satu yang diberi tanda prima. Kemudian untuk setiap prima yang ditemukan, setiap sel array pada index kelipatan dari bilangan tersebut akan ditandai sebagai bukan prima. Pada akhir proses, jika suatu index masih memiliki penanda prima pada sel array yang ditunjuk berarti index tersebut bilangan prima. Cara ini jelas membutuhkan ruang memori untuk mewakili setiap sel array yang dimaksud, namun cara ini akan memangkas tes modulo yang sangat memakan kinerja hardware.

3. 2. Alternatif Dua

Berbeda dengan alternatif sebelumnya, secara umum alternatif ini hanya akan membangkitkan bilangan acak dan menguji sifat primanya dan berhenti apabila bilangan tersebut diyakini prima. Alternatif ini tidak menjamin 100% bilangan tersebut adalah bilangan prima, tetapi menjamin dengan tingkat kesalahan yang relatif kecil sesuai dengan banyaknya proses tes yang dilakukan.

Metode yang cukup sering dipakai dalam pengujian adalah algoritma Lehman yang membagi suatu bilangan yang akan diuji (misal p) dengan bilangan prima kurang dari 256 pengujian dengan cara membangkitkan bilangan acak a yang lebih kecil dari p dan dihitung $a^{(p-1)/2} \bmod p$ yang apabila bernilai 1 atau -1 berarti p berpeluang prima sebesar 50% yang apabila langkah ini diulang dan lolos sebanyak t kali maka akan menghasilkan sebuah bilangan prima p yang mempunyai kesalahan tidak lebih dari $1/2^t$.

Selain dengan metode Lehmann, masih banyak metode lain yang sejenis seperti algoritma Rabin-Miller.

4. PENGUJIAN, HASIL DAN ANALISIS

4. 1. Pengujian dan Hasil

a. Pentingnya sifat prima pada RSA

   Bila pembangkitan pasangan kunci algoritma RSA di bagian 2.4 diikuti, bilangan prima menjadi penentu jalan tidaknya algoritma RSA tersebut. Penggantian peubah prima p dan q pada proses ini meskipun pada akhirnya dapat menghasilkan peubah n, e, dan d, tetap saja peubah tersebut tidak dapat memenuhi persamaan (10).

   Sementara pada proses pembangkitan pasangan kunci algoritma RSA secara umum di bagian 2.4 diawali dengan memilih dua buah bilangan prima. Bila persamaan-persamaan sebelumnya yaitu persamaan (1) hingga (10) diperhatikan kembali, tampak sama sekali tidak terpengaruh oleh dua buah bilangan prima.

Persamaan (5) dan (6) yang menentukan pasangan kunci enkripsi/dekripsi algoritma RSA tidak bergantung pada bilangan prima. Misalkan dipilih sebuah n yang tidak memenuhi $\phi(n) = (p - 1)(q - 1)$ untuk p dan q bilangan prima, yaitu $n = 21$ (n ganjil sehingga tidak mungkin berasal dari p dan q yang genap dimana p dan q harus prima). Secara manual diperoleh $\phi(n) = 12$. Dengan cara manual juga, pasangan yang mungkin menjadi e dan d dan memenuhi persamaan (6) adalah 5 dan 17.

Dengan menggunakan peubah $n = 21$, $e = 5$, dan $d = 17$, telah diuji dapat memenuhi persamaan (10) untuk setiap m anggota bilangan bulat yang lebih kecil dari 21.

b. Pentingnya sifat relatif prima pada RSA

Pada proses pembangkitan pasangan kunci algoritma RSA secara umum di bagian 2.4.d dinyatakan bahwa e relatif prima terhadap $\phi(n)$, penggantian e dengan sembarang bilangan dapat berujung pada tidak dapat ditemukannya d serta proses yang akan menghabiskan lebih banyak waktu untuk mencari kombinasi pasangan e dan d, hal ini dikarenakan pada persamaan (6) jelas pasangan e dan d merupakan pasangan modulo invers terhadap m sehingga supaya kunci e memiliki modulo invers terhadap m, PBB(e, m) harus bernilai 1, begitu juga untuk d.

c. Pembangkitan bilangan prima dengan *brute force*

Berikut deskripsi lingkungan pengujian

CPU	Intel(R) Core(TM)2 Duo T5870 @2.00GHz
Memory	1GB
Operating System	Ubuntu 10.04, Kernel Linux 2.6.32-21-generic
Bahasa Pemrograman	C

Pada lingkungan pengujian tersebut, untuk membangkitkan bilangan prima <= 10.000.000 diperlukan waktu sekitar 20 detik.

d. Pembangkitan bilangan prima dengan mengurangi pembagi dan hanya menguji angka 2 dan bilangan ganjil.

Seperti halnya pengujian pada poin c, lingkungan pengujian pada poin d ini adalah sama, namun waktu yang diperlukan jauh lebih sedikit yaitu sekitar 3 detik.

e. Pembangkitan bilangan prima dengan pendekatan Teori Sieve.

Pada pengujian dengan algoritma ini, waktu yang dibutuhkan jauh lebih sedikit lagi yaitu 0,727 detik pada lingkungan pengujian yang sama.

f. Pembangkitan bilangan prima dengan algoritma Lehman.

Berbeda dengan pengujian sebelumnya, pengujian pada poin ini dilakukan dengan menggunakan bahasa pemrograman Java mengingat library yang telah disediakan. Adapun waktu yang dibutuhkan tidak mencapai 1 detik, namun pada pengujian ini sayangnya tidak bisa menangkap dengan tepat waktu yang dibutuhkan.

4. 2. Analisis

Pengujian a menunjukkan bahwa algoritma RSA sebenarnya tidak bergantung pada bilangan prima, namun penggunaan bilangan prima jauh mempermudah proses perhitungan $\phi(n)$ untuk $n = p \cdot q$ dimana p dan q bilangan prima bila dibandingkan dengan proses perhitungan $\phi(n)$ untuk sembarang bilangan bulat positif n. Hal ini dikarenakan apabila menggunakan bilangan prima maka $\phi(n)$ dapat dengan mudah dihitung menggunakan Fungsi Euler dan Teorema Fermat menjadi $\phi(n) = (p - 1)(q - 1)$ dan tentunya persamaan ini tidak akan dapat dipenuhi jika salah satu dari p atau q tidak prima karena menyebabkan $\phi(p) < (p - 1)$ atau $\phi(q) < (q - 1)$

Pengujian b menunjukkan bahwa sifat relatif prima ini adalah sebuah konsekuensi dari persamaan (6) yang mana e dan d saling modulo invers terhadap m.

Pengujian c, d, dan e, hanya menunjukkan perbandingan yang harus dibayar antara keterbatasan ruang memori dan keterbatasan waktu eksekusi. Sementara f menunjukkan bahwa kecepatan waktu dan sedikitnya ruang memori yang dihabiskan tidak menjamin hasil yang diinginkan yang dalam hal ini adalah kepastian sifat prima dari suatu bilangan. Namun, sangat disayangkan lingkungan pengujian belum mampu menunjukkan ruang memori yang digunakan sehingga kurang representatif dalam melihat perbandingan pertukaran ruang memori dengan waktu eksekusi.

5. KESIMPULAN

Melihat bagaimana algoritma RSA diturunkan, dapat disimpulkan pada dasarnya bilangan prima tidak mutlak harus digunakan, namun penggunaan bilangan prima jauh sangat mempermudah pembangkitan kunci untuk algoritma RSA.

Meskipun terkesan sederhana dan tidak menghabiskan ruang memori dan waktu eksekusi, pembangkitan bilangan prima dengan cara yang tidak menjamin kepastian sifat bilangan prima sebaiknya dihindari bila akan digunakan untuk membangkitkan pasangan kunci enkripsi/dekripsi algoritma RSA. Namun, cara ini tetap dapat dipakai dengan cara memastikan terlebih dahulu pasangan kunci yang dihasilkan apakah memenuhi persamaan (10) atau tidak.

Tidak ada salahnya menunggu lama dan menggunakan sumber daya yang besar untuk menghasilkan bilangan prima sebagai pembangkit pasangan kunci algoritma RSA yang kuat, mengingat sebenarnya pasangan kunci ini tidak terlalu diperlukan banyak pada satu personal atau afiliasi.

REFERENSI

[1] Schneier, Bruce. *Applied Cryptography*. 1996. John Wiley & Sons, Inc.

[2] http://mathworld.wolfram.com/

[3] http://www.rsa.com/

[4] http://www.troubleshooters.com/codecorn/primenumbers/

[5] http://www2.hawaii.edu/~wes/ICS623/

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 17 Mei 2010

ttd

Kamal Mahmudi

13507111

alt1.c

```
#include "alt1.h"

void freePrimeList(struct primeList *head) {
    struct primeList *thisPrime = head;
    struct primeList *nextPrime = thisPrime->next;
    while (nextPrime != NULL) {
        free(thisPrime);
        thisPrime = nextPrime;
        nextPrime = nextPrime->next;
    }
    free(thisPrime);
}

int primeBF(int n) {
    int divisor, prime;
    int i = 2;
    int retval = 0;

    while (i < n) {
        divisor = 2;
        prime = 1;

        while (divisor * divisor <= i && prime) {
            if (i % divisor == 0) prime = 0;
            divisor++;
        }

        if (prime) retval++;
        i++;
    }

    return retval;
}

int primeLD(int n) {
    int prime;
    int i = 2;
    int retval = 0;
    struct primeList *firstPrime = malloc(sizeof (struct primeList));
    assert(firstPrime != NULL);
    struct primeList *latestPrime = firstPrime;
    struct primeList *thisPrime;

    while (i < n) {
        thisPrime = firstPrime;
        prime = 1;

        while (thisPrime != NULL && thisPrime->info * thisPrime->info <= i && prime) {
            if (thisPrime->info > 0 && i % thisPrime->info == 0) prime = 0;
            thisPrime = thisPrime->next;
        }

        if (prime) {
            latestPrime->next = malloc(sizeof (struct primeList));
            assert(latestPrime->next != NULL);
            latestPrime = latestPrime->next;
            latestPrime->info = i;
            latestPrime->next = NULL;
            retval++;
        }

        if (i == 2) i++;
        else i = i + 2;
    }

    freePrimeList(firstPrime);
    return retval;
}

int primeST(int n) {
    int retval = 0;
    char *flag = malloc(sizeof (unsigned char) * (n + 1));
```

```
    assert(flag != NULL);

    int i = 0;
    for (i = 0; i <= n + 1; i++) *(flag + i) = 1;
    flag[0] = 0;
    flag[1] = 0;

    int thisFactor = 2;
    int lastSquare = 0;
    int thisSquare = 0;
    int mark;
    while (thisFactor * thisFactor <= n) {
        mark = thisFactor + thisFactor;
        while (mark <= n) {
            *(flag + mark) = 0;
            mark += thisFactor;
        }

        thisSquare = thisFactor * thisFactor;
        for (; lastSquare < thisSquare; lastSquare++) {
            if (*(flag + lastSquare)) retval++;
        }

        thisFactor++;
        while (*(flag + thisFactor) == 0) thisFactor++;
        assert(thisFactor <= n);
    }

    for (; lastSquare <= n; lastSquare++)
        if (*(flag + lastSquare)) retval++;

    free(flag);
    return retval;
}
```