Bahan Kuliah ke-20

IF5054 Kriptografi

Algoritma Pendukung Kriptografi

Disusun oleh:

Ir. Rinaldi Munir, M.T.

Departemen Teknik Informatika
Institut Teknologi Bandung
2004

20. Algoritma Pendukung Kriptografi

20.1 Algoritma untuk Perpangkatan-Modulo

- Algoritma kunci-publik seperti *RSA*, *ElGamal*, *DSA*, dan sebagainya, sederhana perhitungannya namun sulit implementasinya di dalam perangkat lunak. Hal ini karena algoritma tersebut melakukan operasi perpangkatan dengan bilangan yang besar.

- Misalnya, pada algoritma RSA, setiap blok x_i dienkripsi menjadi blok y_i dengan rumus

  $$y_i = x_i^{\,e} \bmod r$$

  dan blok cipherteks y_i didekripsi kembali menjadi blok x_i dengan rumus

  $$x_i = y_i^{\,d} \bmod r$$

  Pada algoritma *DSA*, kunci publik y dihitung dengan rumus

  $$y = g^x \bmod p$$

  dan sidik dijital dihitung dengan rumus

  $$r = (g^k \bmod p) \bmod q$$

- Karena bilangan bulat yang dioperasikan bisa sangat besar, maka kita perlu membuat algoritma perpangkatan semangkus mungkin.

- Sebagai contoh, kita akan menghitung 5^{123} mod 713. Jika dilakukan secara konvensional, maka

  5^{123} mod 713 = $(5 \times 5 \times \ldots \times 5)$ mod 713
  = 9.40395480657830006374989229777 8e+85 mod 713
  = 435

  Algoritma konvensional (dalam notasi Bahasa *Pascal*) untuk menghitung a^n mod m:

```
function Expo1(a, n, m : LongInt):LongInt
{ Mengembalikan a^n mod m  }
var
   i : integer;
   H : LongInt;
begin
   H:=1;
   for i:=1 to n do
    H:=H*a;
   {endfor}
   Expo1:= H mod m;  { return value }
end;
```

- Dengan algoritma `Expo1` di atas, dibutuhkan *n* kali operasi perkalian dalam perpangkatannya. Untuk n yang besar, algoritma membutuhkan waktu yang lebih lama. Selain itu, nilai antara yang dihasilkan selama perkalian meningkat tajam, sehingga ada kemungkinan tipe *integer* yang digunakan tidak sanggup menampunya.

- Kesulitan di atas dapat diatasi dengan mengingat bahwa

  $$ab \bmod m = [(a \bmod m)(b \bmod m)] \bmod m \qquad (20.1)$$

Contoh 20.1. Sebagai ilustrasi, untuk menghitung 572^{37} mod 713 dapat dilakukan sebagai berikut:

$572^{37} = 572^{36} \cdot 572 = 572^{32} \cdot 572^4 \cdot 572$

$572^2 \bmod 713 = 327184 \bmod 713 = 630$

$572^4 \bmod 713 = 572^2 \,.\, 572^2 \bmod 713$
$= [(572^2 \bmod 713)(572^2 \bmod 713)] \bmod 713$
$= 630^2 \bmod 713 = 396900 \bmod 713 = 472$

$572^8 \bmod 713 = 572^4 \,.\, 572^4 \bmod 713$
$= [(572^4 \bmod 713)(572^4 \bmod 713)] \bmod 713$
$= 472^2 \bmod 713 = 222784 \bmod 713 = 328$

$572^{16} \bmod 713 = 572^8 \,.\, 572^8 \bmod 713$
$= [(572^8 \bmod 713)(572^8 \bmod 713)] \bmod 713$
$= 328^2 \bmod 713 = 107584 \bmod 713 = 634$

$572^{32} \bmod 713 = 572^{16} \,.\, 572^{16} \bmod 713$
$= [(572^{16} \bmod 713)(572^{16} \bmod 713)] \bmod 713$
$= 634^2 \bmod 713 = 401956 \bmod 713 = 537$

$572^{36} \bmod 713 = 572^{32} \,.\, 572^4 \bmod 713$
$= [(572^{32} \bmod 713)(572^4 \bmod 713)] \bmod 713$
$= 537 \,.\, 472 \bmod 713 = 253464 \bmod 713 = 349$

$572^{37} \bmod 713 = 572^{36} \,.\, 572 \bmod 713$
$= [(572^{36} \bmod 713)(572 \bmod 713)] \bmod 713$
$= 349 \,.\, 572 \bmod 713 = 199628 \bmod 713 = 701$

Jadi, $572^{37} \bmod 713 = 701$

- Teknik *divide and conquer* dapat digunakan untuk membagi dua pemangkatnya sampai berukuran kecil. Misalnya,

$$\begin{aligned} a^8 \bmod m &= (a^4 \cdot a^4) \bmod m \\ &= ((a^4 \bmod m)(a^4 \bmod m)) \bmod m \\ &= (a^4 \bmod m)^2 \bmod m \\ &= (a^2 \cdot a^2 \bmod m)^2 \bmod m \\ &= ((a^2 \bmod m)^2 \bmod m)^2 \bmod m \end{aligned}$$

  atau dengan kata lain,

  $a^8 \bmod m = ((a^2)^2)^2 \bmod m = ((a^2 \bmod m)^2 \bmod m)^2 \bmod m$

  Contoh-contoh lainnya,

$$\begin{aligned} a^{16} \bmod m &= (((a^2)^2)^2)^2 \bmod m \\ &= (((a^2 \bmod m)^2 \bmod m)^2 \bmod m)^2 \bmod m \end{aligned}$$

  { hanya membutuhkan 4 operasi perkalian }

$$\begin{aligned} a^{25} \bmod m &= (a^{12})^2 \cdot a \bmod m \\ &= ((a^6)^2)^2 \cdot a \bmod m \\ &= (((a^3)^2)^2)^2 \cdot a \bmod m \\ &= (((a^2 \cdot a)^2)^2)^2 \cdot a \bmod m \\ &= (((((((a^2 \bmod m) \cdot a)^2 \bmod m)^2 \bmod m)^2 \bmod m)^2 \bmod m) \cdot a) \bmod m \end{aligned}$$

  { hanya membutuhkan 7 operasi perkalian }

- Metode perhitungan $a^n \bmod m$ seperti pada teknik *divide and conquer* di atas disebut juga metode *addition chaining* karena hasil perkalian antara langsung dirangkai dengan operasi modulo. Dengan teknik ini, hasil antara tidak akan mencapai bilangan yang besar.

  Algoritma *addition chaining* dengan teknik *divide and conquer* untuk menghitung $a^n \bmod m$:

```
function Expo2(a, n, m : LongInt):LongInt
{ Mengembalikan a^n mod m  }
var
   i : integer;
   H : LongInt;
begin
   if n = 0 then
     Expo2:=1
   else
     if odd(n) then      { n ganjil }
        Expo2:=SQR(Expo2(a, n div 2, p))*a mod m
     else
        Expo2:=SQR(Expo2(a, n div 2, p)) mod m;
    {endif}
  {endif}
end;
```

- Metode *addition chaining* dapat diterapkan secara biner sehingga disebut juga metode *binary square*. Dalam hal ini, pemangkat (*n*) diubah ke dalam bentuk biner baru kemudian dioperasikan.
  Algoritmanya adalah sebagai berikut:

```
function Expo3(a, n, m : LongInt):LongInt
{ Mengembalikan nilai a^n mod m  }
var
   i : integer;
   H : LongInt;
Begin
   ConvertToBiner(n, b); { Konversi n ke dalam biner,
            misalkan bit-bitnya disimpan di dalam b }
   H:=1;
   for i:=1 to Length(b) do
    begin
      H := H*H mod m;
      if b[i] = 1 then
         H:=(H*a) mod m
      {endif}
    end; {for}
   Expo3:=H;
end;
```

Misalkan algoritma Expo3 akan digunakan untuk menghitung

$$2^{129} \bmod 29$$

maka 129 diubah dulu ke dalam biner menjadi 10000001.

Tabel berikut memperlihatkan hasil perhitungan dengan algoritma Expo3:

i	1	2	3	4	5	6	7	8
$b[i]$	1	0	0	0	0	0	0	1
H	2	4	16	24	25	16	24	21
$\sum\times$	2	1	1	1	1	1	1	2

Ket: $\sum \times$ menyatakan jumlah operasi perkalian

Dengan algoritma Expo3, maka perhitungan $2^{129} \bmod 29$ hanya membutuhkan 10 operasi perkalian dan hasil antara tidak mencapai bilangan yang besar sebab hasil antara langsung di-modulo-kan dengan m.

20.2 Tipe Data Bilangan Bulat yang Besar

- Masalah lain yang muncul adalah penggunaan bilangan bulat yang sangat besar. Nilai-nilai parameter di dalam algoritma kriptografi kunci-publik (seperti bilangan prima *p* dan *q*) disarankan adalah bilangan bulat yang sangat besar (panjang). Semua bahasa pemrograman tidak menyediakan tipe data bilangan bulat yang panjang.

- Satu cara untuk mengatasi tipe data tersebut adalah membuat primitif aritmetika bilangan bulat yang besar. Kita dapat menggunakan tipe *string* untuk menyimpan bilangan bulat yang setiap angkanya diperlakukan sebagai karakter. Misalnya, bilangan

  456873245990987645124589O

  akan disimpan sebagai *string*

  ‘456873245990987645124589O’

- Selanjutnya, kita harus membuat primitif operasi aritmetika seperti $a - b$, $a + b$, $a \times b$, a div b, a mod b, dan a^b mod c.

20.3 Pembangkitan Bilangan Prima

- Sebagian besar algoritma kunci-publik menggunakan bilangan prima sebagai salah satu nilai parameternya. Bilangan prima yang disarankan berkuran besar sehingga penggunaan tipe data bilangan bulat yang besar mutlak diperlukan.

- Algoritma untuk membangkitkan bilangan prima adalah:
  1. Bangkitkan bilangan acak p sepanjang n angka.
  2. Uji *brute force* terhadap p dengan membagi p dengan bilangan prima kurang dari 256. Pengujian ini akan menghilangan bilangan bukan prima sebesar 80%. Yang paling mangkus adalah membagi p dengan bilangan prima yang lebih kecil dari 2000.
  3. Jika p berhasil melewati uji *brute force*, uji p dengan algoritma Lehmann.

Algoritma Lehmann

{ Masukan: p (yang akan diuji keprimaannya)
Keluaran: p adalah prima atau tidak prima }

(a) Bangkitkan bilangan acak a yang lebih kecil dari p.
(b) Hitung $a^{(p-1)/2} \bmod p$.
(c) Jika $a^{(p-1)/2} \not\equiv 1$ atau $-1 \pmod p$, maka p tidak prima.
(d) Jika $a^{(p-1)/2} \equiv 1$ atau $-1 \pmod p$, maka peluang p bukan prima adalah 50%.

4. Ulangi pengujian dengan algoritma Lehmann di atas sebanyak t kali (dengan nilai a yang berbeda). Jika hasil perhitungan langkah (b) sama dengan 1 atau -1, tetapi tidak selalu sama dengan 1, maka peluang p adalah prima mempunyai kesalahan tidak lebih dari $1/2^t$.

- Bilangan acak a yang digunakan pada algoritma Lehmann dapat dipilih nilai yang kecil agar perhitungan lebih cepat. Sedangkan jumlah pengujian yang disarankan adalah lima kali.

- Selain algoritma Lehmann, metode lain yang banyak digunakan untuk menguji bilangan prima adalah Rabin-Miller.

Algoritma Rabin-Miller

{ Sebelum algoritma ini dijalankan, lakukan prosedur berikut:
1. Bangkitkan bilangan p yang akan diuji keprimaannya.
2. Hitung b, yang dalam hal ini 2^b adalah nilai pangkat 2 terbesar yang habis membagi $p - 1$.
3. Hitung m sedemikian sehingga $p = 1 + 2^b m$.

Masukan: p, m, dan b
Keluaran: p adalah prima atau tidak prima. }

(a) Bangkitkan bilangan acak a yang lebih kecil dari p.
(b) Nyatakan $j = 0$ dan hitung $z = a^m \bmod p$.
(c) Jika $z = 1$ atau $z = p - 1$, maka p lolos dari pengujian dan mungkin prima.
(d) Jika $z > 0$ dan $z = 1$, maka p bukan prima.
(e) Nyatakan $j = j + 1$. Jika $j < b$ dan $z \neq p - 1$, nyatakan $z = z^2 \bmod p$ dan kembali ke langkah (d). Jika $z = p - 1$, maka p lolos pengujian dan mungkin prima.
(f) Jika $j = b$ dan $z \neq p - 1$, maka p tidak prima.

- Ulangi pengujian dengan algoritma Rabin-Miller di atas sebanyak t kali (dengan nilai a yang berbeda).

4. Tabel Bilangan Prima dari 2 – 256

2	3	5	7	11	13	17	19	23	29
31	41	43	47	53	59	61	67	71	73
79	83	89	97	101	103	107	109	113	127
131	139	149	151	157	163	167	173	179	181
191	193	199	211	223	227	229	233	239	241
251									