STUDI DAN IMPLEMENTASI PENGEMBANGAN ALGORITMA *SUPERINCREASING KNAPSACK* MENJADI ALGORITMA *KNAPSACK* NORMAL

Bayu Adi Persada – NIM : 13505043

Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
E-mail : if15043@students.if.itb.ac.id

Abstrak

Algoritma *knapsack* merupakan salah satu dari algoritma pembangkitan kunci publik. Secara harfiah, *knapsack* dapat diartikan sebagai tas, kantung, atau karung. Permasalahan utama dalam *knapsack* adalah optimalisasi kombinatorial karena sebuah tas atau kantung memiliki kapasitas yang terbatas dalam memuat benda-benda yang dimasukkan. Oleh karena itu, algoritma *knapsack* ini memiliki keunggulan dari segi keamanan dalam sulitnya memecahkan persoalan optimalisasi kombinatorial atau disebut juga *knapsack problem*.

Superincreasing knapsack merupakan modifikasi *knapsack* yang dapat dipecahkan dalam orde polinomial *O(n)*. Kekurangan dari algoritma *superincreasing knapsack* ini adalah mudah untuk dipecahkan sehingga dari segi keamanan, algoritma ini bukan merupakan algoritma kriptografi yang kuat.

Algoritma *superincreasing knapsack* adalah algoritma yang lemah karena cipherteks dapat didekripsi dalam waktu lanjar untuk mendapatkan plainteks awal. Namun, algoritma *knapsack* normal atau *non-superincreasing knapsack* memiliki kompleksitas yang lebih rumit terutama dalam segi komputasi karena membutuhkan dalam orde eksponensial untuk memecahkannya. Maka dari itu, algoritma *superincreasing knapsack* yang dimodifikasi menjadi algoritma *knapsack* normal dengan memanfaatkan kunci publik untuk enkripsi dan kunci privat untuk dekripsi. Pemanfaatan ini membuat kunci publik sebagai barisan *non-superincreasing* dan kunci privat sebagai barisan *superincreasing*.

Kata kunci: *knapsack*, algoritma *superincreasing knapsack*, algoritma *knapsack* normal, *superincreasing*, *non-superincreasing*, kunci publik, kunci privat.

1. Pendahuluan

Permasalahan dalam algoritma *knapsack* adalah bagaimana menempatkan benda-benda agar dapat dimasukkan ke dalam suatu tempat dengan kapasitas tertentu secara maksimal. Benda-benda tersebut sebanyak n ($I_1,\ldots,I_n$) akan di masukkan ke dalam sebuah karung dengan kapasitas C. Jika setiap benda I_j memiliki beban w_j dan keuntungan b_j maka setiap benda tersebut akan dimasukkan ke dalam karung dengan tujuan tidak melebihi kapasitas beban karung tersebut dan mendapati keuntungan yang maksimal.

Secara umum, permasalahan *knapsack* dapat dituliskan dalam bentuk:
$C = I_1+I_2+I_3+\ldots I_n$, atau
$C = b_1w_1+b_2w_2+b_3w_3+\ldots b_nw_n$.
Di dalam teori algoritma, persoalan *knapsack* termasuk ke dalam kelompok *NP-complete*, yaitu persoalan yang tidak dapat dipecahkan dalam orde waktu polinomial.

Ide dasar dari sebuah algoritma kriptografi *knapsack* sederhana adalah menyandikan pesan untuk menjadi rangkaian solusi dari persoalan *knapsack*. Setiap beban w_i di dalam persoalan *knapsack* merupakan kunci privat, sedangkan bit-bit plainteks dinyatakan sebagai b_j.

Contoh:
Misalkan terdapat 5 buah benda yang akan dimasukkan ke dalam *knapsack* dengan masing-masing beban benda tersebut adalah 2, 5, 9, 13, dan 17. Kemudian, misalkan terdapat sebuah plainteks 11011001100101110011 01101 yang akan dienkripsi menggunakan algoritma *knapsack*.

Plainteks tersebut dibagi menjadi blok yang sesuai dengan banyaknya objek benda, yaitu 5. Setelah itu, setiap bit dalam blok dikalikan

dengan beban benda (w_i) yang berkorespondensi sesuai.

Blok 1 : 11011
Knapsack : 2,5,9,13,17
Kriptogram : 2+5+0+13+17 = 37

Blok 2 : 00110
Knapsack : 2,5,9,13,17
Kriptogram : 0+0+9+13+0 = 22

Blok 3 : 01011
Knapsack : 2,5,9,13,17
Kriptogram : 0+5+0+13+17 = 35

Blok 4 : 10011
Knapsack : 2,5,9,13,17
Kriptogram : 2+0+0+13+17 = 32
Blok 5 : 01101
Knapsack : 2,5,9,13,17
Kriptogram : 0+5+9+0+17 = 31

Jadi, cipherteks yang dihasilkan adalah (37, 22, 35, 32, 31).

Kekurangan algoritma *knapsack* sederhana tersebut adalah tidak dapat digunakan untuk mendekripsi cipherteks yang dihasilkan sehingga didapatkan kembali pesan awal. Pada contoh di atas, jika diberikan kriptogram 37, maka akan ditentukan b_1, b_2, b_3, b_4, b_5 sedemikian sehingga

$$37 = 2b_1 + 5b_2 + 9b_3 + 13b_4 + 17b_5.$$

Solusi persamaan di atas tersebut tidak dapat dipecahkan dalam orde waktu polinomial dengan semakin besarnya *n* (dengan asumsi barisan beban tidak cenderung berurut naik). Namun, di sisi lain, hal tersebut juga dapat dijadikan kekuatan algoritma *knapsack*.

2. *Superincreasing Knapsack*

Secara teori, *superincreasing knapsack* adalah persoalan *knapsack* yang dapat dipecahkan dalam orde polinomial atau *O(n)*. Namun, *superincreasing knapsack* merupakan persoalan *knapsack* yang mudah dipecahkan sehingga bukan merupakan algoritma kriptografi yang kuat dan kokoh.

Barisan *superincreasing* adalah suatu barisan di mana setiap nilai di dalam barisan lebih besar dari pada jumlah semua nilai sebelumnya. Sebagai contoh, {2, 4, 7, 13, 28, 55} merupakan barisan *superincreasing* sedangkan {2, 4, 5, 9, 25, 34} bukan barisan *superincreasing*. Jadi, dapat disimpulkan bahwa sebuah *knapsack* $C = (I_1, I_2, I_3, \ldots, I_n)$ dikatakan *superincreasing* jika

$$I_j > \sum_{i=1}^{j-1} I_i \qquad \text{untuk } 2 \le j \le n.$$

Solusi dari *superincreasing knapsack* dapat dijabarkan dalam langkah-langkah sebagai berikut:

a. Jumlahkan semua beban atau bobot di dalam barisan.
b. Bandingkan beban total dengan bobot terbesar di dalam barisan tersebut. Jika beban terbesar lebih kecil atau sama dengan beban total, maka benda tersebut dimasukkan ke dalam *knapsack*. Jika lebih besar, maka benda tersebut tidak dapat dimasukkan.
c. Kurangi beban total dengan beban yang telah dimasukkan. Setelah itu, bandingkan kembali beban total sekarang dengan beban terbesar selanjutnya. Langkah tersebut akan terus dilakukan sampai seluruh beban di dalam barisan selesai dibandingkan.
d. Jika beban total telah menjadi nol, maka terdapat solusi dari persoalan *superincreasing knapsack*. Akan tetapi, jika nilainya tidak nol, maka tidak terdapat solusi untuk permasalahan tersebut.

Dalam algoritma tersebut, *T* merupakan beban total. Iterasi akan terus dilakukan hingga menemukan solusi, yaitu hingga beban total *T* telah bernilai nol dan terdapat himpunan solusi *knapsack*.

Seperti yang telah dijelaskan pada paragraf-paragraf sebelumnya, *superincreasing knapsack* relatif mudah untuk didekripsi atau dengan kata lain algoritma tersebut tidak memproteksi pesan. Hal ini dikarenakan orang lain akan dapat mengetahui pola bit dari beban total untuk *superincreasing knapsack* jika elemen-elemen dari *superincreasing knapsack* diketahui.

Contoh:

Terdapat barisan *superincreasing* adalah (2,3,6,13,27,52) dan beban total atau kapasitas *knapsack* (*W*) sebesar 70 sehingga di dapat persamaan

$$70 = 2b_1+3b_2+6b_3+13b_4+27b_5+52b_6.$$

Untuk mendapatkan b_1, b_2, …, b_6 dilakukan proses sebagai berikut:

$W = 70 > w_6 = 52$; w_6 dimasukkan;
$W\text{-}w_6 = 18 < w_5 = 27$; w_5 tidak dimasukkan;
$W\text{-}w_6 = 18 > w_4 = 13$; w_4 dimasukkan;
$W\text{-}w_6\text{-}w_4 = 5 < w_3 = 6$; w_3 tidak dimasukkan;
$W\text{-}w_6\text{-}w_4 = 5 > w_2 = 3$; w_2 dimasukkan;
$W\text{-}w_6\text{-}w_4\text{-}w_2 = 2 \le w_1 = 2$; w_1 dimasukkan;
$W=0$.

Karena beban total atau kapasitas *knapsack* tidak bersisa (0), maka solusi persoalan *superincreasing knapsack* ditemukan. Barisan beban yang dimasukkan ke *knapsack* adalah (2,3,13,52).

3. Pengembangan Algoritma *Superincreasing Knapsack*

Algoritma *knapsack* normal atau *non-superincreasing knapsack* adalah algoritma yang komputasinya sangat sulit. Oleh karena itu, pemecahannya pun cukup sulit dengan membutuhkan waktu dalam orde eksponensial (O^n).

Namun, algoritma *superincreasing knapsack* dapat dimodifikasi menjadi algoritma *knapsack* normal dengan memanfaatkan metode yang diperkenalkan oleh Martin Hellman dan Ralph Merkle. Algoritma tersebut diberi nama Merkle-Hellman.

Algoritma Merkle-Hellman memanfaatkan kunci publik dan kunci privat untuk memodifikasi *superincreasing knapsack* menjadi *knapsack* normal. Kunci publik yang berbentuk barisan *non-superincreasing* dimanfaatkan untuk enkripsi pesan sedangkan untuk dekripsi pesan, digunakan kunci privat yang merupakan barisan *superincreasing*.

Secara sistematis, algoritma Merkle-Hellman untuk membangkitkan kunci publik dan kunci privat dideskripsikan sebagai berikut:

a. Tentukan himpunan angka yang merupakan barisan *superincreasing*.
b. Kalikan setiap elemen di dalam barisan tersebut dengan *r* modulo *q*. Modulus *q* harus bernilai lebih besar daripada jumlah semua elemen di dalam barisan *superincreasing* tersebut sedangkan nilai *r* seharusnya tidak memiliki faktor persekutuan dengan *q*.
c. Hasil perkalian akan menjadi kunci publik sedangkan barisan *superincreasing* semula akan dijadikan kunci privat.

3.1 Sistem Merkle-Hellman

Sistem Merkle-Hellman mengenkripsi *n*-bit pesan $m = (\alpha_1,\ldots,\alpha_n) \in M$ dengan menggunakan kunci publik $e = (\beta_1,\ldots,\beta_n)$ di mana $\alpha_1 \in \{0,1\}$ dan $\beta_1 \in Z_q$; $i = 1,\ldots, n$ dan q adalah bilangan prima.
Kriptogram $c \in C$ dikalkulasikan sebagai

$$c = \sum_{i=1}^{n} \alpha_i \beta_i . \qquad \textbf{(1)}$$

Pada tahap pertama, kunci publik dan elemen-elemen rahasia ditentukan dan dibuat oleh penerima pesan. Dalam sistem Merkle-Hellman ini, penerima pesan adalah pihak yang mengatur keseluruhan sistem.
Pihak penerima ini lah yang pertama kali menentukan barisan *superincreasing* $w = (w_1, \ldots, w_n)$ dimana

$$w_i > \sum_{j=1}^{i-1} w_j . \qquad \textbf{(2)}$$

Yang perlu menjadi catatan adalah *w* merupakan *superincreasing knapsack* yang dapat diselesaikan dalam waktu linier.

Setelah itu, pihak penerima menentukan Z_q dengan *q* merupakan bilangan prima dan sebuah pengali $r \in Z_q$. Kedua bilangan prima, *q* dan *r*, dapat dipilih secara acak menurut

$$q > \sum_{i=1}^{n} w_i .$$

Pada tahap selanjutnya, penerima mentransformasikan vektor *superincreasing knapsack w* menurut persamaan:

$$\beta_i \equiv w_i \, r \pmod{q} \qquad \textbf{(3)}$$

untuk $i = 1, \ldots, n$.

Deretan $(\beta_1, \ldots, \beta_n)$ merupakan kunci publik dari sistem. Hal penting pada aplikasi sistem Merkle-Hellman adalah nilai vector *w*, pengali *w*, dan bilangan prima *q* akan tetap dirahasiakan oleh penerima.

Jika penerima menerima pesan terenkripsi atau kriptogram *c* yang dibuat menurut persamaan (1), maka penerima pesan dapat mengkonversikan pesan tersebut dengan

$$c' \equiv c \, r^{-1} \pmod{q} .$$

Dengan memanfaatkan persamaan (1) dan (3), maka didapatkan

$$c' \equiv \sum_{i=1}^{n} \alpha_i \beta_i r^{-1} \equiv \sum_{i=1}^{n} \alpha_i w_i \pmod{q} .$$

Nilai kriptogram yang telah ditransformasikan *c'* berkorespondensi pada *superincreasing knapsack* yang dapat dipecahkan dalam orde suku banyak (polinomial) sehingga penerima dapat mengetahui bit-bit α_I dari pesan *m*.

Contoh:
Untuk mengilustrasikan sistem Merkle-Hellman, anggap 5 bit pesan dikirimkan ke pihak penerima. Seperti yang telah dijelaskan sebelumnya, pihak penerima menginisiasi algoritma ini dengan menentukan vektor,

$w = (w_1, w_2, w_3, w_4, w_5) = (2,3,6,12,25)$.

Sebagai syarat barisan *superincreasing*, perlu dicatat bahwa:

$$w_2 > w_1$$
$$w_3 > w_1 + w_2$$
$$w_4 > w_1 + w_2 + w_3$$
$$w_5 > w_1 + w_2 + w_3 + w_4.$$

Selanjutnya dilakukan pemilihan nilai pasangan bilangan (r,q) secara acak sesuai dengan aturan bahwa q adalah bilangan prima dan $q > \sum_{i=1}^{5} w_i$ yang bernilai 48. Ambil nilai q = 53 dan nilai r = 46. Sangat mudah untuk mengetahui bahwa r^{-1} = 15 (mod 53).

Kemudian, pihak penerima akan mengkalkulasikan nilai kunci publik dengan menggunakan persamaan (3), yaitu

$$\beta_1 \equiv w_1.r \pmod{q} \equiv 39 \pmod{53}$$
$$\beta_2 \equiv w_2.r \pmod{q} \equiv 32 \pmod{53}$$
$$\beta_3 \equiv w_3.r \pmod{q} \equiv 11 \pmod{53}$$
$$\beta_4 \equiv w_4.r \pmod{q} \equiv 22 \pmod{53}$$
$$\beta_5 \equiv w_5.r \pmod{q} \equiv 37 \pmod{53}.$$

Jadi, dapat disimpulkan bahwa kunci publik $e = (\beta_1, \beta_2, \beta_3, \beta_4, \beta_5)$ = (39, 32, 11, 22, 37) dikirim ke pihak pengirim. Kemudian, dapat dianggap bahwa pihak penerima telah menerima kriptogram atau pesan terenkripsi dengan c = 119. Untuk mendekripsi pesan tersebut, transformasi dilakukan sebagai berikut:

$c' = c.r^{-1}$ = 119 . 15 (mod 53)= 36 (mod 53).

Tahap selanjutnya adalah memecahkan persoalan *knapsack* dengan cara sebagai berikut:

c' = 36 > w_5 = 25; sehingga nilai α_5 = 1
$c'-w_5$ = 11 < w_4 =12 ; nilai α_4 = 0
$c'-w_5$ = 11 > w_3 =6 ; nilai α_3 = 1
$c'- w_5- w_3$ = 5 > w_2 =3 ; nilai α_2 = 1
$c'-w_5- w_3- w_2$ = 2 < w_1 =2 ; nilai α_1 = 1.

Jadi, pihak penerima telah mengkonversikan pesan m = (1, 1, 1, 0, 1).

3.2 Implementasi Sistem Merkle-Hellman

Sistem Merkle-Hellman berikut ini akan diimplementasikan dalam bahasa Java dengan nama **MerkleHellman.java**. Kelas yang akan dibuat bernama MerkleHellman. Dalam implementasi sistem ini, terdapat empat proses utama, yaitu mengambil sebuah file teks, membuat kunci publik, enkripsi plainteks, dan dekripsi cipherteks.

```
import java.lang.*;
import java.math.*;
import java.util.*;
import java.io.*;
import java.security.*;
import java.math.BigInteger;

public class MerkleHellman
{
    public static final boolean
        Success     = true;
    public static final boolean Fail
        = true;
    public HexRead hr = new
        HexRead();
    public byte[] plaintext;
    String toBin = new String();
    public BigInteger[] a ;
    public BigInteger[] b;
    BigInteger[] x;
    BigInteger[] decodedtext;
    BigInteger m,k,criptotext;

    public MerkleHellman() {
        loadFile();
        genKey();
        encrypt();
        decrypt();
    }

    public void loadFile() {
        try{
        plaintext = new
        byte[hr.readData("Test.txt")
        .length];

        plaintext =
        hr.readData("Test.txt").clon
        e();
        }

        catch(IOException e)
        {}
        BigInteger tobin = new
        BigInteger(plaintext);
        toBin = tobin.toString(2);

        System.out.println("M = " +
        toBin);

        a = new
        BigInteger[toBin.length()];

        for(int
        i=0;i<toBin.length();i++)
        {
           a[i] = new BigInteger("2");
           a[i]=a[i].pow(i);
        }

        System.out.print("Barisan
        superincreasing = ");
        for(int
        i=0;i<toBin.length();i++)

        System.out.print(a[i].intValue
        () + ",");
        System.out.println();
    }
```

```
public void genKey() {
     m = new BigInteger("0");

     for(int i=0;i<a.length;i++)
         m = m.add(a[i]);
     System.out.println("modulus" +
     m.toString());

     k =m.subtract(BigInteger.ONE);

     System.out.println("W" +
     k.toString());

     b = new BigInteger[a.length];

     for(int i =0;i<a.length;i++)
     {
      b[i]=k.multiply(a[i]).mod(m);
     }

     System.out.print("Kunci = ");

     for(int i=0;i<b.length;i++)
     System.out.print(b[i].intValue
     () + ",");
}

public void encrypt() {
     criptotext = new
     BigInteger("0");
     for(int i=0;i<b.length;i++)
     {
     System.out.println("$"+Charact
     er.toString(toBin.charAt(i))+"
     *"+b[i].longValue());

     long num =
     Long.parseLong(Character.toStr
     ing(toBin.charAt(i)))*b[i].lon
     gValue();

     criptotext =
     criptotext.add(BigInteger.valu
     eOf(num));
     }

     System.out.println();

     System.out.println("Cipherteks
     - "+criptotext);
 }

 public boolean SiSSP(BigInteger s) {
     BigInteger s1=s;
     x = new
     BigInteger[toBin.length()];

     for (int i=toBin.length()-1;
     i>=0; i--)
       if (s1.compareTo(a[i])==1 ||
       s1.compareTo(a[i])==0)
       {
          s1 = s1.subtract(a[i]);
          x[i] = new
                 BigInteger("1");
       }
       else x[i] = new
       BigInteger("0");

     BigInteger sum = new
     BigInteger("0");

     for (int i=0; i<toBin.length();
     i++) sum =
     sum.add(x[i].multiply(a[i]));
       if (sum == s) return true;
          return false;
 }
```

```
    public void decrypt()
    {
        BigInteger s,c,w;
        s = k.modInverse(m);

        System.out.println("s =
        "+s.multiply(criptotext).mod(m).
        toString());

        decodedtext = new
        BigInteger[toBin.length()];

        if
        (SiSSP(s.multiply(criptotext).mo
        d(m)) )
           decodedtext[i] = x[i];

        System.out.println("<<<<Hasil
        Dekripsi>>>> ");
        System.out.print("M = ");

        for (int i=0; i<toBin.length();
        i++)
        System.out.print(decodedtext[i]);
     }

      public static void main(String[]
        args) throws Exception
     {
        MerkleHellman mh = new
        MerkleHellman();
     }
}
```

Penjelasan:

```
public HexRead hr = new HexRead();
```

Metode `HexRead` membaca *hex data* dari sebuah file dan akan memberikan keluaran berupa byte[].

```
BigInteger tobin = new
        BigInteger(plaintext);
toBin = tobin.toString(2);
```

BigInteger `tobin` tersebut nantinya akan diubah ke dalam string biner dengan basis 2.

```
for(int i=0;i<toBin.length();i++)
   {
    a[i] = new BigInteger("2");
    a[i]=a[i].pow(i);
   }
```

Proses di atas bertujuan untuk membentuk barisan *superincreasing.*

```
k =m.subtract(BigInteger.ONE);
```

Perlakuan di atas untuk membuat k relatif prima terhadap m.

```
b = new BigInteger[a.length];
for(int i =0;i<a.length;i++)
{
   b[i]=k.multiply(a[i]).mod(m);
}
```

Proses tersebut di atas dibuat sesuai dengan persamaan (3) untuk membuat kunci publik.

```
for(int i=0;i<b.length;i++)
        {
        System.out.println("$"+Charact
        er.toString(toBin.charAt(i))+"
        *"+b[i].longValue());

        long num =
        Long.parseLong(Character.toStr
        ing(toBin.charAt(i)))*b[i].lon
        gValue();

        criptotext =
        criptotext.add(BigInteger.valueO
        f(num));
        }
```

Proses di atas dilakukan pada proses enkripsi dengan cara yang sama untuk memecahkan persoalan *knapsack*, yaitu mengalikan bit plainteks dengan kunci publik yang berkorespondensi.

```
for (int i=toBin.length()-1;  i>=0; i--
)
   if (s1.compareTo(a[i])==1 ||
       s1.compareTo(a[i])==0)
      {
         s1 = s1.subtract(a[i]);
         x[i] = new BigInteger("1");
      }
   else x[i] = new BigInteger("0");
```

Proses di atas terdapat pada fungsi SiSSP yang bertujuan untuk memecahkan masalah *superincreasing.* Fungsi ini akan digunakan dalam proses dekripsi. Fungsi tersebut menggunakan fungsi bawaan `compareTo` yang mengembalikan -1 jika nilai a[i] lebih kecil dari s1, 0 jika nilainya sama, dan 1 jika nilai a[i] lebih besar dari s1.

```
s = k.modInverse(m);

System.out.println("s =
"+s.multiply(criptotext).mod(m).toStrin
g());

decodedtext = new
BigInteger[toBin.length()];

if
(SiSSP(s.multiply(criptotext).mod(m)))
        for (int i=0; i<toBin.length();
        i++)
                decodedtext[i] = x[i];
```

Proses di atas berperan pada saat dekripsi pesan. Sesuai dengan $c' \equiv c.r^{-1} (\mathrm{mod}\, q)$, maka akan didapatkan plainteks dengan mengalikan kriptogram dengan r^{-1} mod q, kemudian nyatakan hasil kalinya sebagai penjumlahan elemen-elemen kunci privat.

4. Kesimpulan dan Saran

4.1 Kesimpulan

Algoritma *knapsack* merupakan algoritma pembangkitan kunci publik yang cukup aman dinilai dari kesulitan untuk memecahkan persoalan *knapsack*.

Algortima *knapsack* sederhana hanya dirancang untuk mengenkripsi sebuah pesan namun tidak dapat digunakan untuk mendekripsi cipherteks yang dihasilkan. Permasalahan algoritma ini tidak dapat diselesaikan dalam waktu orde suku banyak (polinomial).

Algoritma *superincreasing knapsack* merupakan algoritma yang kurang cocok untuk diimplementasikan sebagai algoritma kriptografi karena sifatnya yang mudah dipecahkan dalam orde polinomial *O(n)*.

Walaupun algoritma *superincreasing knapsack* adalah algoritma yang lemah, namun terdapat keuntungan yang tidak dimiliki algoritma *knapsack* sederhana, yaitu cipherteks dapat didekripsi kembali menjadi sebuah plainteks dengan mudah.

Oleh karena itu, dalam pembangkitan kunci publik, algoritma *superincreasing knapsack* dimodifikasi menjadi algoritma *knapsack* normal atau non-*superincreasing knapsack* dengan memanfaatkan barisan non-*superincreasing* dan *superincreasing* untuk dijadikan kunci publik dan kunci privat. Algoritma ini diberi nama Sistem Merkle-Hellman.

Enkripsi pada Sistem Merkle-Hellman dilakukan dengan cara yang sama seperti algoritma *knapsack*. Setelah plainteks dipecah menjadi blok bit yang panjangnya sama dengan kardinalitas barisan kunci publik, setiap bit dalam blok tersebut dikalikan dengan elemen yang berkoresponden di dalam kunci publik. Untuk melalukan dekripsi, dimanfaatkan kunci privat yang merupakan barisan *superincreasing*.

4.2 Saran

Untuk menambah kekuatan algoritma *knapsack*, kunci publik maupun kunci privat sebaiknya terdiri dari paling sedikit 250 elemen. Selain itu, nilai dari setiap elemen memiliki panjang antara 200 dan 400 bit [RIN06]. Dengan nilai-nilai sepanjang itu, diperkirakan dibutuhkan waktu selama 10^{46} tahun untuk memecahkan kunci tersebut secara *brute force*, dengan asumsi satu juta percobaan tiap detik.

Sistem Merkle-Hellman dapat dipecahkan oleh Shamir pada tahun 1984 yang mempresentasikan algoritma waktu polinomial yang mengkalkulasikan *knapsack* yang mudah dari kunci publik. Shamir menggunakan properti *superincreasing* dari bilangan integer sebuah *knapsack* untuk memperoleh ketidaksamaan linier dari sistem. Oleh karena itu, untuk meningkatkan kompleksitas algoritma Merkle-Hellman ini, Shamir merumuskan sebuah transformasi untuk merekonstruksi *knapsack* normal menjadi *superincreasing knapsack.*

5. Daftar Pustaka

[MER78] Merkle, Ralph, Martin Hellman. 1978. *Hiding Information and Signatures in Trapdoor Knapsacks.* IEEE Trans Information Theory, pp525-530.

[MOR90] Mortello, Silvano, Paolo Toth. 1990. *Knapsack Problem.* John Wiley & Sons, England.

[MUN06] Munir, Rinaldi. 2006. *Kriptografi.* Penerbit Informatika, Bandung.

[SHA82] Shamir, Adi. 1982. *A Polynomial Time Algorithm for Breaking the Basic Merkle-Hellman Cryptosystem.* CRYPTO, pp279-288.