PERBANDINGAN ALGORITMA MD2, MD4, DAN MD5

Muhammad Bahari Ilmy – NIM : 13504062

Program Studi Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
E-mail : bahari135@students.itb.ac.id

Abstrak

Makalah ini membahas tentang algoritma hash *MD2, MD4,* dan *MD5*. Ketika varian algoritma ini merupakan varian yang berkembang dari hasil pengembangan dari varian sebelumnya. Ketiga algoritma fungsi hash ini dikembangkan oleh Ron Rivest di MIT untuk *RSA Data Security*. MD5 saat ini telah dikenal sebagai algoritma yang cukup banyak dipakai secara luas pada dunia kriptografi sebagai salah satu pilihan algoritma fungsi hash.

Perbedaan yang dibahas dalam makalah ini dari segi kompleksitas, performansi, dan tingkat keamanan masing-masing algoritma. Sebenarnya ketiga algoritma ini merupakan algoritma yang cukup mirip, karena ketiganya dibuat oleh orang yang sama. Selanjutnya algoritma yang lebih baru merupakan perbaikan dari algoritma sebelumnya. Untuk tingkat kemanan, akan dibahas seberapa jauh kriptanalis dapat menembus dan menemukan kolisi untuk masing-masing algoritma.

Sebuah perangkat lunak digunakan untuk mencoba bermacam-macam variasi input untuk algoritma *MD2*, *MD4*, dan *MD5* tersebut.

***Kata kunci**: MD2, MD4, MD5,* fungsi hash, *collision, message digest*

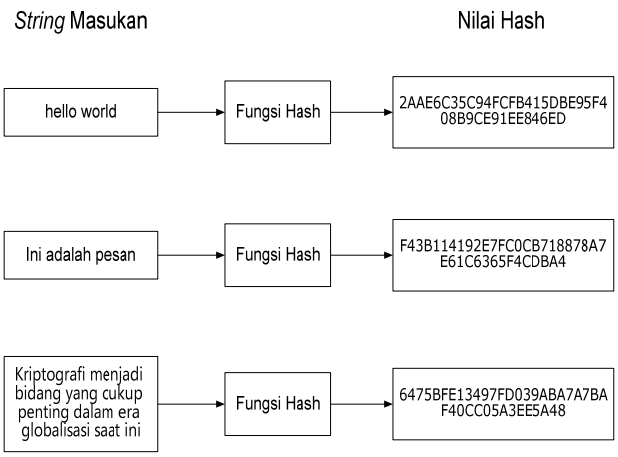
1. Pendahuluan

Dalam dunia kriptografi terdapat sebuah fungsi yang sesuai digunakan untuk aplikasi keamanan seperti otentikasi dan integritas pesan. Fungsi tersebut adalah fungsi hash. Sebuah algorima fungsi hash atau yang biasa disebut juga sebagai *message digest algorithm* beroprasi pada sebuah input string dengan ukuran yang bervariasi dan menghasilkan sebuah output string yang bersifat unik dengan ukuran yang tetap (pada umumnya berukuran jauh lebih kecil dari pada ukuran semula). Output string ini kemudian biasa disebut dengan nilai hash atau *message digest*. Sebenarnya fungsi hash ini banyak dikembangkan dewasa ini salah satunya adalah untuk digunakan pada proses penandatangan digital dokumen, namun fungsi hash sebenarnya dapat digunakan secara luas dalam beragam aplikasi.

Fungsi hash dapat menerima masukan *string* apa saja. Jika *string* menyatakan pesan atau *message* makan sembarang pesan M berukuran bebasa dikompresi oleh fungsi hash H melalui persamaan :

$$h = H(M)$$

Pada persamaan di atas, h adalah nilai hash atau *message digest* dari fungsi H untuk masukan string M. Gambar 1 memperlihatkan contoh 3 buah pesan dengan panjang berbeda-beda selalu menghasilkan sebuah nilai hash yang panjangnya tetap (pada contoh ini dihasilkan sebuah nilai hash yang memiliki panjang 160 bit yang dinyatakan dalam kode heksadesimal. Satu karakter heksdesimal = 4 bit). Nama lain fungsi hash adalah : fungsi kompresi/kontraksi (*compression function*), cetak-jari(*fingerprint*), *cryptographic checksum, message digest check (MIC)*, *manipulation detection code (MDC)*.



Gambar 1 – Contoh hashing beberapa buah pesan dengan panjang yang berbeda-beda

Salah satu aplikasi fungsi *hash* diantaranya adalah untuk memverifikasi kesamaan salinan suatu arip aslinya yang tersimpan di dalam sebuah basis data terpusat. Daripada mengirim salinan arsip tersebut secara keseluruhan ke komputer pusat yang biasanya membutuhkan waktu transmisi yang cukup lama dan ongkos yang mahal lebih mangkus jika yang dikirimkan hanya *message digest*-nya saja. Jika *message-digest* untuk salinan arsip sama dengan *message-digest* arsip asli, berarti salinan arsip tersebut sama dengan arsip di dalam basis data.

2. Sifat Fungsi *Hash*

Fungsi *hash* didesain dengan beberapa sifat yang menempel padanya. Terdapat tiga sifat yang penting berada pada sebuah algoritma fungsi *hash*. Sebuah algoritma fungsi *hash* yang baik harus memiliki ketiga sifat ini.

Sifat pertama yaitu bila diberikan sebuah nilai *output* atau yang biasa disebut dengan nilai *hash* atau *message digest* dari sebuah fungsi *hash*, maka seharusnya tidak bisa ditemukan *input* yang sesuai berkorespondensi yang dapat membuat nilai *hash* tersebut.

Sifat kedua sebenarnya merupakan implikasi dari sifat yang pertama, yang mana bila diberikan sebuah pasangan *input* dan *output* dari sebuah fungsi *hash,* maka seharusnya seharusnya tidak dapat dihasilkan *input* lain yang dapat menghasilkan nilai *output* yang sama. Kedua sifat ini biasa dikenal dengan sebutan fungsi satu arah (*one-way function*).

Fungsi satu-arah (*one-way function*) adalah fungsi hash yang bekerja dalam satu arah yaitu pesan yang sudah diubah menjadi *message digest* tidak dapat dikembalikan menjadi pesan semula. Dua pesan yang berbeda akan menghasilkan nilai *hash* yang berbeda pula. Sifat-sifat fungsi satu arah diantaranya sebagai berikut :

1. Fungsi *H* dapat diterapkan pada blok data dengan ukuran berapa saja.
2. *H* menghasilkan nilai (h) dengan panjang tetap (*fixed-lenghth output*).
3. *H(x)* mudah dihitung untuk setiap nilai x yang diberikan.
4. Untuk setiap *h* yang diberikan, tidak mungkin menemukan sedemikian sehingga *H(x) = h*. Itu sebabnya fungsi *H* dikatakan fungsi *hash* satu-arah (*one-way hash function*).
5. Untuk setiap x yang diberikan, tidak mungkin mencari pasangan y ≠ x sedemikian sehingga *H(y) = H(x)*.
6. Tidak mungkin secara komputasi mencari pasangan x dan y sedemikian sehingga *H(x) = H(y)*.

Sifat-sifat di atas sangat penting untuk sebuah fungsi *hash*, sebab sebuah fungsi *hash* seharusnya berlaku seperti fungsi acak. Sebab fungsi *hash* dianggap tidak aman jika (i) secara komputasi dimungkinkan menemukan pesan yang bersesuaian dengan pesan ringkasnya (*message digest*), dan (ii) terjadi kolisi (*collision*), yaitu terdapat beberapa pesan berbeda yang mempunyai pesan ringkas yang sama.

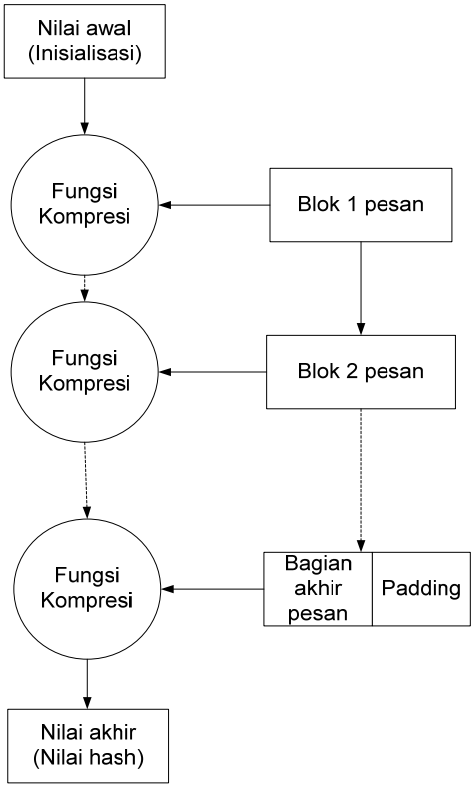
3. Struktur Fungsi Hash

Kebanyakan fungsi hash memiliki kemiripan dalam struktur iterasi. Struktur tersebut berlandaskan pada kaidah-kaidah pada fungsi kompresi. Pendek kata, komputasi untuk menghasilkan fungsi hash sangat bergantung pada pesan *input* yang diberikan (*message depend*) yang biasa disebut dengan variabel berantai *(chaining variable).*

Pada permulaan proses *hashing, chaining variable* tersebut diinisialisasi dengan suatu nilai tertentu yang sifatnya tetap. Ha ini merupakan salah satu dari spesifikasi algoritma fungsi *hash*.

Lalu fungsi kompresi digunakan untuk mengubah atau *update* nilai-nilai yang terdapat *chaining variable* dengan menggunakan cara atau algoritma yang cukup rumit. Proses ini berlanjut secara rekursif dengan *chaining variable* selalu di-update terus menerus untuk setiap bagian pada pesan tersebut, hingga semua pesan telah digunakan. Nilai akhir dari *chaining variable* akan menjadi nilai hash yang berkorespondensi dengan pesan tersebut.

Secara umum struktur pada algoritma fungsi hash dapat digambarkan seperti pada gambar berikut ini :



Gambar 2 – Stuktur umum algoritma hash

4. Algoritma MD2

Algortma MD2 dikembangkan oleh Ron Rivest pada tahun 1989. Algoritma ini dioptimasikan dengan mennggunakan komputer 8-bit. MD2 sebenarnya dispesifikasikan dalam RFC 1319.

Algoritma MD2 menghasilkan nilai *hash* yang berukuran 128-bit dan menerima input pesan dengan panjang yang tidak ditentukan. Pesan yang akan dijadikan input untuk fungsi *hash* ini akan terlebih dahulu akan dipadding. Untuk kalkulasi yang sebenarnya.

Misalkan kita memiliki sejumlah b-byte pesan sebagai input, dan kita mengharapkan untuk dapat mendapatkan *message digest* dari pesan tersebut. Dalam hal ini, b merupakan suatu bilangan bulat sembarang yang bernilai positif, bisa juga nol, dan besarnya sembarang. Kita nyatakan bahwa byte dari pesan ditulis dalam bentuk :

m_0 m_1 . . . m_{b-1}

Di bawah ini akan dijelaskan 5 tahap proses untuk menghasilkan *message digest* untuk algoritma *MD2*.

4.1 Memasukkan *Padding Byte*

Pesan akan ditambahkan melalui proses *padding* sehingga panjang pesan tersebut kongruen dengan 0 modulo 16. Maka, pesan akan diperluas sehingga panjangnya merupakan kelipatan dari 16 byte. Proses *padding* ini selalu dilakukan meskipun panjang pesan awal sebelum dilakukan *padding* sudah kongruen dengan 0 modulo 16.

Padding dilakukan dengan mengikuti : "*i*" byte dari nilai "*i*" akan ditambahkan pada pesan sehingga panjang pesan kongruen dengan 0 modulo 16. Maka ukuran *padding byte* paling sedikit 1 byte sampai 16 byte.

Pada tahap ini pesan hasil *padding* memiliki panjang pesan kelipatan 16 byte. Kita bagi pesan menjadi M[0 . . . N-1] dimana N merupakan kelipatan 16.

4.2 Memasukkan *Checksum*

Sebanyak 16 byte *checksum* dari pesan akan ditambahkan pada hasil dari tahap sebelumnya. Pada langkah ini digunakan sebuah 256-byte yang dibangkitkan secara acak yang dibuat dengan nilai digit dari pi. Jika S[i] menotasikan untuk elemen ke-i pada tabel.

Pseudokode untuk checksum

```
/* kosongkan checksum. */
For i = 0 to 15 do:
  Set C[i] to 0
end
/* dari loop i */

Set L to 0

/* Proses tiap blok 16-word.
*/
For i = 0 to N/16-1 do

   /* Checksum block i. */
   For j = 0 to 15 do
     Set c to M[i*16+j]
     Set C[j] to S[c xor L]
     Set L to C[j]
   end
end
```

Selanjutnya 16-byte checksum tersebut dimasukkan ke dalam pesan.

4.3 Inisialisasi Penyangga *MD*

Sebuah 48-byte penyangga X digunakan uttuk menghasilkan *message digest*, nilai penyangga diinisialisasi dengan nol.

4.4 Proses pesan dalam blok 16-byte

Langkah ini menggunakan angka hasil pemnbangkitan sebanya 256-byte yang sama yang dihasilkan pada proses 4.2.

Pseudokode :

```
/* Proses tiap blok 16-word */
For i = 0 to N'/16-1 do

  /* Menyalin blok i pada X */
  For j = 0 to 15 do
    Set X[16+j] to M[i*16+j]
    Set X[32+j] to (X[16+j]
xor
    X[j])
  end /* dari loop j */

  Set t to 0

  /* Lakukan 18 round */
  For j = 0 to 17 do

    /* Round j */
    For k = 0 to 47 do
      Set t and X[k] to (X[k]
xor
      S[t]).
    end /* dari loop k */

    Set t to (t+j) modulo 256.
  end /* dari loop j */

end /* dari loop i */
```

5. Algoritma MD4

MD4 didesain untuk sebagai algoritma fungsi *hash* yang memiliki kemangkusan dalam segi waktu. *MD4* dibuat oleh Ronald Rivest pada Oktober 1990.

Pertama-tama kita misalkan pesan memiliki sejumlah b-bit pesan sebagai input, dan kita menginginkan untuk menghasilkan *message digest* dari pesan tersebut. Dalam hal ini, b merupakan sebuah bilangan bulat positif, mungkin nol, dan bilangan tersebut harus kelipatan dari 8. Kita membagi pesan tersebut sebagai berikut :

m_0 m_1 . . . m_{b-1}

Di bawah ini akan dijelaskan lima tahap proses menghasilkan *message digest* untuk algoritma *MD4*.

5.1 Memasukkan *Padding Bit*

Pertama-tama dilakukan *padding* pada pesan awal sehingga panjangnya (dalam satuan bit) kongruen dengan 448 modulo 512. Maka pesan tersebut kekurangan 64 bit untuk mencapai kelipatan 512. *Padding* selalu dilakukan sehingga pesan tersebut kongruen dengan 512.

Padding bit awal yaitu bit "1", selanjutnya ditambahkan bit "0" sehingga pesan tersebut memiliki panjang yang kongruen dengan 448 modulo 512. Jadi, panjang *padding* paling sedikit adalah satu bit hingga 512 bit.

5.2 Memasukkan Panjang Pesan

Sebuah 64-bit yang direpresentasikan oleh b (panjang pesan awal sebelum dilakukan *padding bit*) dimasukkan pada hasil pesan untuk tahap satu di atas. Panjang pesan yang digunakan selalu lebih kecil dari 2^{64}. Bila panjang pesan melebihi 2^{64}, maka order terendah yang direpresentasikan oleh 2^{64} yang akan digunakan. Oleh karena itu, setelah proses ini, pesan akan memiliki panjang kelipatan dari 512 bit. Dan pesan dibagi dalam kelipatan 32 bit dalam M[0 . . . N-1] yang menotasikan pesan, dimana N adalah kelipatan dari 16.

5.3 Inisialisasi Penyangga

Ada empat peubah penyangga yang digunakan, yaitu *A, B, C, D* yang digunakan untuk menghasilkan *message digest*. Masing-masing variable ini merepresentasikan sebuah nilai 32-bit register. Register tersebut diinisialisasi dengan nilai dalam bentuk heksadesimal, dalam order terendah terlebih dahulu) :

```
A = 01 23 45 67
B = 89 ab cd ef
C = fe dc ba 98
D = 76 54 32 10
```

5.4 Proses Pesan Dalam Blok 32-bit

Untuk tahap ini pertama-tama kita definisikan terlebih dahulu fungsi-fungsi pelengkap yang dibutuhkan untuk menghasilkan *message digest*, Terdapat tiga fungsi pelengkap yang dalam hal ini tiap-tiap fungsi ini menerima input 32-bit dan menghasilkan output 32 bit juga.

$$F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z)$$
$$G(X,Y,Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$$
$$H(X,Y,Z) = X \oplus Y \oplus Z$$

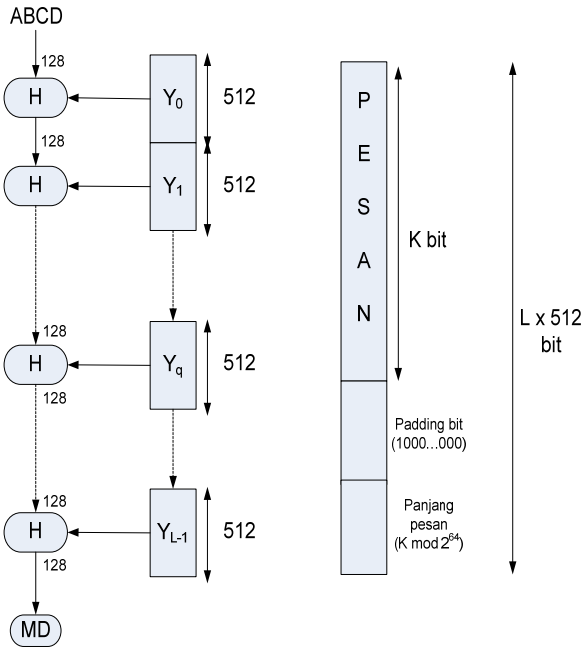
Dalam tiap bit posisi F pada fungsi berlaku hubungan kondisional berikut : if X then Y else Z. Fungsi F seharusnya didefinisikan menggunakan atau tanpa v selama XY dan not(X)Z tidak memiliki bit “1” pada posisi yang sama. Dalam tiap bit posisi G berlaku hubungan berikut : jika paling tidak dua dari X, Y, Z sedang digunakan, maka G memiliki sebuah bit “1” dalam posisi tersebut, dan jika tidak G memilii sebuah bit “0”. Ini menarik untuk diperhatikan bahwa jika bit-bit X, Y, dan Z saling bebas tidak saling mempengaruhi satu sama lain, tiap-tiap bit dari f(X,Y,Z) akan menjadi saling bebas pula. Fungsi H merupakan operasi bit XOR atau fungsi *parity*. Dia memiliki atribut yang mirip dengan F dan G.

```
/* Proses tiap 32-bit pesan */
      For i = 0 to N/16-1 do
        /* Salin blok i pada X
*/
        For j = 0 to 15 do
          Set X[j] to
M[i*16+j].
        End
/* Simpan A sebagai AA, B
sebagai BB, C sebagai CC, dan
D sebagai DD */
        AA = A
        BB = B
        CC = C
        DD = D
/* Round 1 */
/* [abcd k s] menotasikan
operasi :
a = (a + F(b,c,d) + X[k]) <<<
s. */

/* Lakukan 16 kali operasi */
[ABCD  0  3]  [DABC  1  7]
[CDAB  2 11]  [BCDA  3 19]
[ABCD  4  3]  [DABC  5  7]
[CDAB  6 11]  [BCDA  7 19]
[ABCD  8  3]  [DABC  9  7]
[CDAB 10 11]  [BCDA 11 19]
[ABCD 12  3]  [DABC 13  7]
[CDAB 14 11]  [BCDA 15 19]
/* Round 2 */
/* [abcd k s] menotasikan
operasi
a = (a + G(b,c,d) + X[k] +
5A827999) <<< s. */
/* Lakukan 16 kali operasi */
[ABCD  0  3]  [DABC  4  5]
[CDAB  8  9]  [BCDA 12 13]
[ABCD  1  3]  [DABC  5  5]
[CDAB  9  9]  [BCDA 13 13]
[ABCD  2  3]  [DABC  6  5]
[CDAB 10  9]  [BCDA 14 13]
[ABCD  3  3]  [DABC  7  5]
[CDAB 11  9]  [BCDA 15 13]
```

```
/* Round 3 */
/* [abcd k s] menotasikan
operasi
a = (a + H(b,c,d) + X[k] +
6ED9EBA1) <<< s. */

/* Lakukan 16 kali operasi */
[ABCD  0  3]  [DABC  8  9]
[CDAB  4 11]  [BCDA 12 15]
[ABCD  2  3]  [DABC 10  9]
[CDAB  6 11]  [BCDA 14 15]
[ABCD  1  3]  [DABC  9  9]
[CDAB  5 11]  [BCDA 13 15]
[ABCD  3  3]  [DABC 11  9]
[CDAB  7 11]  [BCDA 15 15]

/* Lalu lakukan penambahan
berikut ini. (Dalam hal ini
increment dilakukan pada tiap-
tiap register) */

        A = A + AA
        B = B + BB
        C = C + CC
        D = D + DD

end
```

6. Algoritma MD5

MD5 dibuat oleh Ronald Rivest pada tahun 1991. MD5 merupakan fungsi *hash* satu arah yang merupakan perbaikan dari *MD4* setelah *MD4* berhasil ditemukan kelemahannya oleh kriptanalis. Algoritma *MD5* menerima masukan berupa pesan dengan ukuran sembarang dan menghasilkan *message digest* yang panjangnya 128 bit. Gambar skema pembuatan *message digest* diperlihatkan pada gambar di bawah ini.



Gambar 3 – Gambar skema proses pembuatan *message digest* dengan algoritma *MD5*

Teradapat 4 langkah pokok dalam proses pembuatan *message digest* pada algoritma MD5, yaitu :

1. Penambahan bit-bit pengganjal (*padding-bit*).
2. Penambahan nilai panjang pesan semula.
3. Inisialisasi penyangga (*buffer*) *MD* sebagai nilai awal.
4. Pengolahan pesan dalam blok 512 bit.

6.1 Penambahan Bit-Bit Pengganjal

Pesan ditambah dengan bit-bit pengganjal hingga panjang pesan (dalam satuan bit) kongruen dengan 448 modulo 512. Hal ini berarti bahwa panjang pesan setelah ditambahi bit-bit pengganjal adalah 64 bit kurang dari kelipatan 512. Angka 512 muncul karena *MD5* memproses pesan dalam blok-blok yang berukuran 512. Pesan dengan pankang 448 bit pun tetap ditambah dengan bit-bit pengganjal. Jika panjang pesan 448 bit, maka pesan tersebut ditambah dengan 512 bit menjadi 960 bit. Jadi, panjang bit-bit pengganjal adalah antara 1 sampai 512. Bit-bit pengganjal terdiri dari sebuah bit 1 diikuti dengan sisanya bit 0.

6.2 Penambahan Nilai Panjang Pesan Semula

Pesan yang telah diberi bit-bit pengganjal selanjutnya ditambah lagi dengan 64 bit yang menyatakan panjang pesan semula. Jika panjang pesan > 2^{64} maka yang diambil adalah panjangnya dalam modulo 2^{64}. Dengan kata lain, jika panjang pesan semula adalah K bit, maka 64 bit yang ditambahkan menyatakan K modulo 2^{64}. Setelah ditambah dengan 64 bit, panjang pesan sekarang menjadi kelipatan 512 bit.

6.3 Inisialisasi Penyangga *MD*

MD5 membutuhkan 4 buah penyangga (*buffer*) yang masing-masing panjangnya 32 bit. Total panjang penyangga adalah 4 x 32 = 128 bit. Keempat penyangga ini menampung hasil antara dan hasil akhir. Keempat penyangga ini diberi nama A, B, C, D. Setiap penyangga diinisialisasi dengan nilai-nilai (dalam notasi HEX) sebagai berikut :

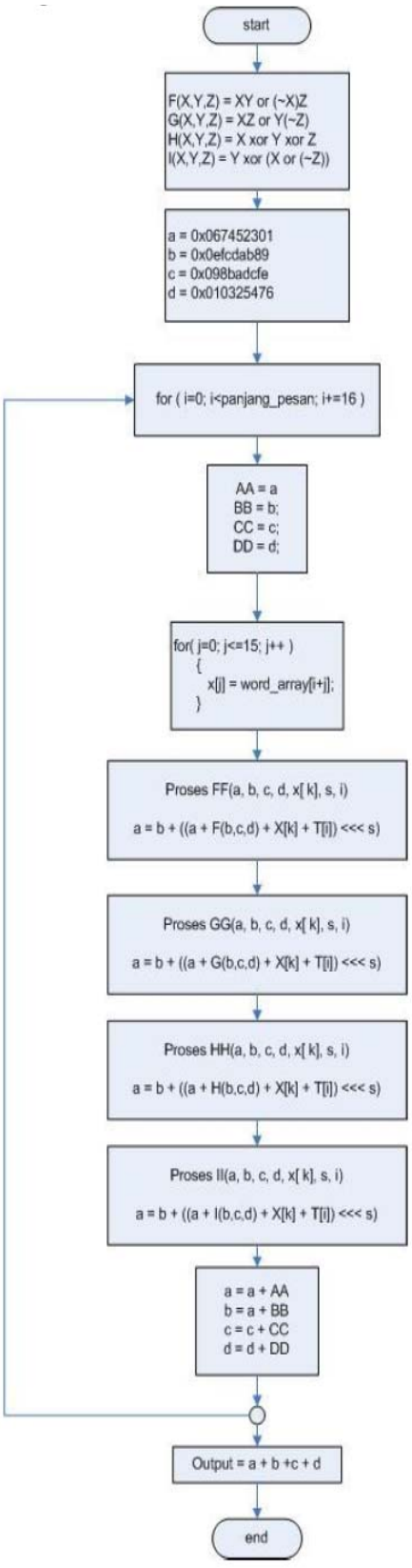
```
A = 01 23 45 67
B = 89 ab cd ef
C = fe dc ba 98
D = 76 54 32 10
```

(Catatan : beberapa versi MD5 menggunakan nilai inisialisasi yang berbeda, yaitu :

```
A = 67 45 23 01
B = EF CD AB 89
C = 98 BA DC FE
D = 10 32 54 67 )
```

6.4 Pengolahan Pesan dalam Blok Berukuran 512 bit

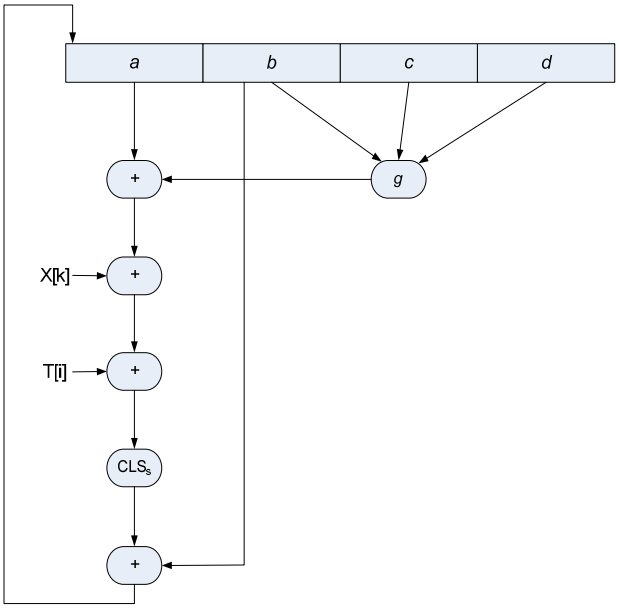
Pesan dibagi menjadi *L* buah blok yang masing-masing panjangnya 512 bit (Y_0 sampai Y_{L-1}). Setiap blok 512-bit diproses bersama dengan penyangga *MD* menjadi keluaran 128-bit, dan ini disebut proses H_{MD5}, Gambar proses H_{MD5} diperlihatkan pada gambar di bawah ini.



Gambar 4 – Proses manipulasi blok 512 bit dalam *MD5*

Proses H terdiri dari 4 buah putaran, dan masing-masing putaran melakukan operasi dasar *MD5* sebanyak 16 kali dan setiap kali operasi dasar memakai sebuah elemen *T*. Jadi setiap putaran mamakai 16 elemen tabel *T*. Pada gambar, Y_q di atas menyatakan blok 512-bit ke-*q* dari pesan yang telah ditambah bit-bit pengganjal dan tambahan 64 bit nilai panjang pesan semula. MD_q adalah nilai *message digest* 128-bit dari proses *H* ke-*q*. Pada awal proses, MD_q berisi nilai inisialisasi penyangga *MD*.

Fungsi-fungsi f_F, f_G, f_H, dan f_I masing-masing berisi 16 kali operasi dasar terhadap masukan, setiap operasi dasar menggunakan elemen tabel T. Operasi dasar *MD5* diperlihatkan pada gambar di bawah ini.



Gambar 5 – Operasi dasar *MD5*

Operasi dasar *MD5* yang diperlihatkan pada gambar di atas daoat ditulis dengan sebuah persamaan sebagaI berikut :

$$a \leftarrow b + \mathrm{CLS}_s(a + g(b,c,d) + X[k] + T[i])$$

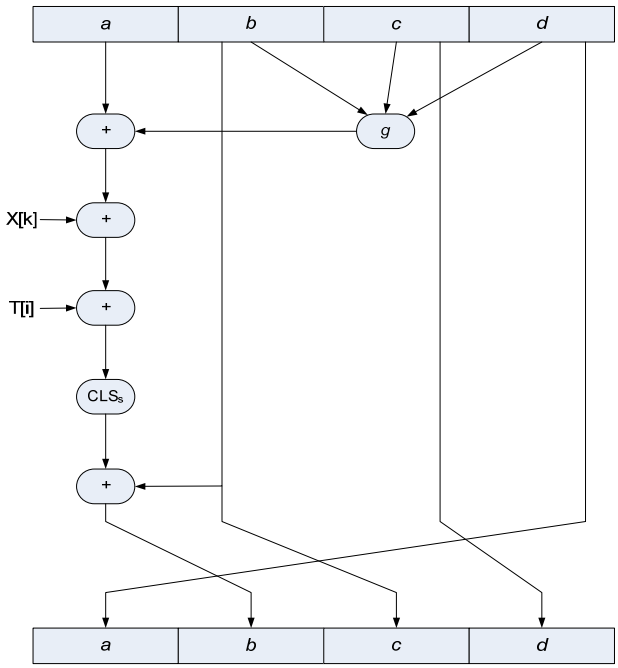
di mana,

- *a, b, c, d* = empat buah variable penyangga 32-bit (berisi nilai penyangga *A, B, C, D*)
- *g* = salah satu fungsi *F, G, H, I*
- CLS_s = *circular left shift* sebanyak *s* bit (notasi : <<< s)
- *X*[*k*] = kelompok 32-bit ke-*k* dari blok 512 bit *message* ke-*q*. Nilai *k* = 0 sampai 15
- *T*[*i*] = elemen Tabel *T* ke-*i* (32 bit)
- + = operasi penjumlahan modulo 2^{32}

Karena ada 16 kali operasi dasar, maka setiap kali selesai satu operasi dasar, penyangga-penyangga tersebut digeser ke kanan secara sirkuler dengan cara pertukaran sebagai berikut :

```
temp ← d
d ← c
c ← b
b ← a
a ← temp
```

Pergeseran tersebut dapat digambarkan pada gambar di bawah, yang dalam hal ini nilai penyangga *a* yang baru disalin ke dalam penyangga *b*, nilai penyangga *b* (yang lama) disalin ke dalam penyangga *c*, nilai penyangga *c* (yang lama) disalin ke dalam penyangga *d*, dan nilai penyangga *d* (yang lama) disalin ke dalam penyangga *a*.



Gambar 6 – Operasi dasar *MD5* dengan pergeseran penyangga ke kanan secara sirkuler

Fungsi f_F, f_G, f_H, dan f_I adalah fungsi untuk memanipulasi *a, b, c* dan *d* dengan ukuran 32-bit. Masing-masing fungsi dapat dilihar pada gambar di bawah ini.

```
F(X,Y,Z) = XY or (~X)Z
G(X,Y,Z) = XZ or Y(~Z)
H(X,Y,Z) = X xor Y xor Z
I(X,Y,Z) = Y xor (X or (~Z))
```

Gambar 7 – Fungsi-fungsi pada MD5

Nilai *T*[*i*] dapat dilihat pada tabel di bawah ini. Tabel tersebut terbentuk dari fungsi $t(i) - 2^{32}$ x abs(sin(*i*)), yang dalam hal ini sudut *i* dalam satuan radian.

Tabel 1 – Nilai *T*[*i*]

T[01]: d7 6a a4 78	T[33]: ff fa 39 42
T[02]: e8 c7 b7 56	T[34]: 87 71 f6 81
T[03]: 24 20 70 db	T[35]: 69 d9 61 22
T[04]: c1 bd ce ee	T[36]: fd e5 38 0c
T[05]: f5 7c 0f af	T[37]: a4 be ea 44
T[06]: 47 87 c6 2a	T[38]: 4b de cf a9
T[07]: a8 30 46 13	T[39]: f6 bb 4b 60
T[08]: fd 46 95 01	T[40]: be bf bc 70
T[09]: 69 80 98 d8	T[41]: 28 9b 7e c6
T[10]: 8b 44 f7 af	T[42]: ea a1 27 fa
T[11]: ff ff 5b b1	T[43]: d4 ef 30 85
T[12]: 89 5c d7 be	T[44]: 04 88 1d 05
T[13]: 6b 90 11 22	T[45]: d9 d4 d0 39
T[14]: fd 98 71 93	T[46]: e6 db 99 e5
T[15]: a6 79 43 8e	T[47]: 1f a2 7c f8
T[16]: 49 b4 08 21	T[48]: c4 ac 56 65
T[17]: f6 1e 25 62	T[49]: f4 29 22 44
T[18]: c0 40 b3 40	T[50]: 43 2a ff 97
T[19]: 26 5e 5a 51	T[51]: ab 94 23 a7
T[20]: e9 b6 c7 aa	T[52]: fc 93 a0 39
T[21]: d6 2f 10 5d	T[53]: 65 5b 59 c3
T[22]: 02 44 14 53	T[54]: 8f 0c cc 92
T[23]: d8 a1 e6 81	T[55]: ff ef f4 7d
T[24]: e7 d3 fb cb	T[56]: 85 84 5d d1
T[25]: 21 e1 cd e6	T[57]: 6f a8 7e 4f
T[26]: c3 37 07 d6	T[58]: fe 2c e6 e0
T[27]: f4 d5 0d 87	T[59]: a3 01 43 14
T[28]: 45 5a 14 ed	T[60]: 4e 08 11 a1
T[29]: a9 e3 e9 05	T[61]: f7 53 7e 82
T[30]: fc ef a3 f8	T[62]: bd 3a f2 35
T[31]: 67 6f 02 d9	T[63]: 2a d7 d2 bb
T[32]: 8d 2a 4c 8a	T[64]: eb 86 d3 91

Dari persamaan di atas dapat dilihat bahwa masing-masing fungsi f_F, f_G, f_H, dan f_I melakukan 16 kali operasi dasar, misalkan notasi

$$[abcd \quad k \quad s \quad i]$$

menyatakan operasi

$$a \leftarrow b+((a+g(b,c\ d) + X[k] + T[i] <<<s))$$

yang dalam hal ini $<<<s$ melambangkan operasi *circular left shift* 32-bit, maka operasi dasar pada masing-masing putaran dapat ditabulasikan sebagai berikut :

Putaran 1 : 16 kali operasi dasar dengan

$$g(b,c,d) = F(b,c,d)$$

diberikan pada tabel dibawah ini

Tabel 2 – Rincian operasi pada fungsi *F*(*b,c,d*)

No	[*abcd*	*k*	*s*	*i*]
1	[ABCD	0	7	1]
2	[DABC	1	12	2]
3	[CDAB	2	17	3]
4	[BCDA	3	22	4]
5	[ABCD	4	7	5]
6	[DABC	5	12	6]
7	[CDAB	6	17	7]
8	[BCDA	7	22	8]
9	[ABCD	8	7	9]
10	[DABC	9	12	10]
11	[CDAB	10	17	11]
12	[BCDA	11	22	12]
13	[ABCD	12	7	13]
14	[DABC	13	12	14]
15	[CDAB	14	17	15]
16	[BCDA	15	22	16]

Putaran 2 : 16 kali operasi dasar dengan

$$g(b,c,d) = G(b,c,d)$$

diberikan pada tabel di bawah ini

Tabel 3 – Rincian operasi pada fungsi *G*(*b,c,d*)

No	[*abcd*	*k*	*s*	*i*]
1	[ABCD	1	5	17]
2	[DABC	6	9	18]
3	[CDAB	11	14	19]
4	[BCDA	0	20	20]
5	[ABCD	5	5	21]
6	[DABC	10	9	22]
7	[CDAB	15	14	23]
8	[BCDA	4	20	24]
9	[ABCD	9	5	25]
10	[DABC	14	9	26]
11	[CDAB	3	14	27]
12	[BCDA	8	20	28]
13	[ABCD	13	5	29]
14	[DABC	2	9	30]
15	[CDAB	7	14	31]
16	[BCDA	12	20	32]

Putaran 3 : 16 kalo operasi dasar dengan

$$g(b,c,d) = H(b,c,d)$$

diberikan pada tabel di bawah ini

Tabel 4 – Rincian operasi pada fungsi *H*(*b,c,d*)

No	[*abcd*	*k*	*s*	*i*]
1	[ABCD	5	4	33]
2	[DABC	8	11	34]
3	[CDAB	11	16	35]
4	[BCDA	14	23	36]
5	[ABCD	1	4	37]
6	[DABC	4	11	38]
7	[CDAB	7	16	39]
8	[BCDA	10	23	40]
9	[ABCD	13	4	41]
10	[DABC	0	11	42]
11	[CDAB	3	16	43]
12	[BCDA	6	23	44]
13	[ABCD	9	4	45]
14	[DABC	12	11	46]
15	[CDAB	15	16	47]
16	[BCDA	2	23	48]

Putaran 4 : 16 kali operasi dengan

$$g(b,c,d) = \mathrm{I}(b,c,d)$$

diberikan pada tabel di bawah ini

Tabel 5 – Rincian operasi pada fungsi *I*(*b,c,d*)

No	[*abcd*	*k*	*s*	*i*]
1	[ABCD	0	6	49]
2	[DABC	7	10	50]
3	[CDAB	14	15	51]
4	[BCDA	5	21	52]
5	[ABCD	12	6	53]
6	[DABC	3	10	54]
7	[CDAB	10	15	55]
8	[BCDA	1	21	56]
9	[ABCD	8	6	57]
10	[DABC	15	10	58]
11	[CDAB	6	15	59]
12	[BCDA	13	21	60]
13	[ABCD	4	6	61]
14	[DABC	11	10	62]
15	[CDAB	2	15	63]
16	[BCDA	9	21	64]

Setelah putaran keempat, *a*, *b*, *c*, dan *d* ditambahkan ke *A, B, C,* dan *D*, dan selanjutnya algoritma memproses untuk blok data berikutnya (Y_{q-1}). Keluaran akhir dari algoritma *MD5* adalah hasil penyambungan bit-bit di *A, B, C,* dan *D*.

Dari uraian di atas, secara umum fungsi *hash MD5* dapat ditulis dalam persamaan matematis berikut ini :

$MD_0 = IV$

$MD_{q+1} = MD_q + f_I(Y_q + f_H(Y_q + f_G(Y_q + f_F(Y_q + MD_q))))$

$MD = MD_{L-1}$

yang dalam hal ini,

IV : *initial vector* dari penyangga *ABCD*, yang dilakukan pada proses inisialisasi penyangga
Y_q : blok pesan berukuran 512-bit ke-*q*
L : jumlah blok pesan
MD : nilai akhir *message digest*

7. Kriptanalisis

7.1 Kriptanalisis terhadap algoritma MD2

Pesan hash yang dihasilkan dengan algoritma *MD2* dilakukan melalui tiga fase yang berbeda sebagaimana telah dijelaskan di atas. Pertama, dilakukan *padding* pada pesan sehingga panjang pesan menjadi kelipatan 16. Fase kedua adalah kita menghitung checksum yang berukuran 16 byte yang kemudian ditambahkan pada pesan yang telah di-*padding* tadi. Checksum merupakan sebuah fungsi yang menerima input pesan dan menghitung di bawah substitusi non-linear dengan menggunaan tabel yang berdasarkan digit dari pi. String hasil tersebut kemudian dibagi menjadi blok-blok yang berukuran 16 byte. Blok terakhir kemudian merupakan checksum sebenarnya. Fase ketiga terdiri dari aplikasi yang mirip sifat fungsi komptersi yang menghitung nilai baru menggunakan rantai variabel sebagai fungsi. Nilai awal dari variabel rantai tersebut telah ditentukan secara spesifik sebagai bagian dari desain algoritma, dan para proses akhir variable rantai tersebut digabungkan dan menjadi 16-byte nilai hash *MD2*.

Kemajuan dalam bidang kriptanalisis khususnya terhadap algoritma ini telah banyak ditunjukan oleh para ahli, bahwa sangat mungkin ditemukan kolisi pada algoritma ini. Tepatnya, sangat mungkin ditemukan dua buah string 16-byte yang dihasilkan ketika inisialisasi variabel awal dengan menggunakan sebuah nilai tertentu yang berbeda sebagai variabel rantai,dan dapat menghasilkan output yang sama. Hal ini tentu akan menuju pada sebuah kolisi pada *MD2* yang mana faktanya bahwa komputasi akhir dari sebuah nilai hash *MD2* tidak lagi bergantung pada nilai dari checksum dan ini dapat dianalogikan sebagai dua buah pesan yang berbeda sebagai input.
Pada saat ini tampaknya sengan sulit untuk membuat kelonggaran terhadap checksum yang telah ada selama proses kriptanalisis. Tanpa menggunakan checksum, *MD2* akan lebih mudah untuk dikriptanalisis.

Hal ini mendekatkan kita pada tingkat kenyamanan dan kewaspadaan tertentu untuk menggunakan algoritma *MD2*. Oleh karena itu, *MD2* sudah lama tidak direkomendasikan untuk digunakan dalam berbagai aplikasi dimana aplikasi tersebut membutuhkan resistansi dari kolisi.

Petanyaan yang muncul adalah tentang kelanjutan dari penggunaan algoritma ini bahwa ternyata masih terdapat aplikasi yang menggunakan algoritma ini sebagai kebutuhan untuk menjadi fungsi hash.

Walaupun begitu, masih sangat sulit untuk menghasilkan *preimage* (input pesan yang diberikan sebagai masukkan untuk fungsi *MD2*) dari sebuah *message digest* tertentu. Namun tidak menutup kemungkinan bahwa kemajuan ini akan membawa kita pada hal tersebut.

7.2 Kriptanalisis terhadap algoritma MD4

MD4 didesain untuk menghasilkan sebuah fungsi hash yang memiliki performansi dalam kecepatan. Kesuksesan dalam kriptanalisis pada algoritma ini terletak dari pengurangan jumlah round pada versi *MD4*. Namun, perancangan algoritma *MD4* direpresentasikan dengan sebuah kompromi antara keamanan dan kecepatan. Sebagai hasilnya *MD4* kurang memiliki tingkat keamanan yang cukup baik.

Tabel 6 – Contoh Kolisi pada MD4

X1	M1	4d7a9c83 56cb927a b9d5a578 57a7a5ee de748a3c dcc366b3 b683a020 3b2a5d9f c69d71b3 f9e99198 d79f805e a63bb2e8 45dd8e31 97e31fe5 2794bf08 b9e8c3e9
	M1’	d11d0b96 9c7b41dc f497d8e4 d555655a c79a7335 cfdebf0 66f12930 8fb109d1 797f2775 eb5cd530 baade822 5c15cc79 ddcb74ed 6dd3c55f d80a9bb1 e3a7cc35
	H1	5f5c1a0d 71b36046 1b5435da 9b0d807a
	H1’	4d7e6a1d efa93d2d de05b45d 864c429b

X2	M2	4d7a9c83 56cb927a b9d5a578 57a7a5ee de748a3c dcc366b3 b683a020 3b2a5d9f c69d71b3 f9e99198 d79f805e a63bb2e8 45dd8e31 97e31fe5 f713c240 a7b8cf69
	M2’	4d7a9c83 d6cb927a 29d5a578 57a7a5ee de748a3c dcc366b3 b683a020 3b2a5d9f c69d71b3 f9e99198 d79f805e a63bb2e8 45dc8e31 97e31fe5 f713c240 a7b8cf69
	H2	e0f76122 c429c56c ebb5e256 b809793

7.3 Kriptanalisis terhadap algoritma MD5

Berbagai macam usaha untuk membuat serangan terhadap algoritma *MD5* telah lebih banyak dibandingkan dengan algoritma *MD2* dan *MD4*. Hal pertama yang perlu diperhatikan dalam kriptanalisis *MD5* adalah ditemukannya apa yang dinamakan *pseudo-collision*. *Pseudo-collision* untuk sebuah fungsi kompresi atau fungsi hash dicontohkan dengan nilai fix dari beberapa blok pesan dan dapat ditemukan dua nilai yang berbeda pada variabel rantai atau penyangga sehingga menghasilkan output yang sama.

Serangan pertama yang dianggap cukup penting yang terjadi pada MD5 adalah penemuan *pseudo-collision2* oleh den Boer dan Bosselaers [5]. Serangan ini diumumkan pada tahun 1993. Dua inisialisasi vektor yang berbeda, memiliki kompresi MD5 yang sama. Kedua vektor ini berbeda 4 bit satu sama lain. Salah satu aspek penting yang perlu diperhatikan adalah, pada MD5, hanya satu variabel (*initial variable*) yang digunakan secara berantai oleh setiap tahapan. *Pseudo-collision* bergerak lebih ke arah penemuan sebuah variabel yang menghasilkan output yang sama untuk dua pesan yang berbeda dalam sebuah fungsi kompresi. Namun variabel ini belum tentu sama dengan *initial variable*. Jika variable yang digunakan sama dengan *initial variable*, maka *collision* yang terjadi dianggap *full collision* terhadap MD5. Pada tahun 1996, Hans Dobbertin dalam publikasinya, memberikan deskripsi yang lebih jelas mengenai kemungkinan terjadinya *collision* pada fungsi kompresi MD5 ini.

Pada bulan Maret 2004, dikembangkan sebuah proyek yang disebut MD5CRK. Proyek ini mencoba membuktikan kelemahan MD5 dengan mencari kemungkinan *collision* dengan cara *brute force*, atau acak. Pertimbangan yang diambil adalah, ukuran dari enkripsi, yaitu sebesar 128 bit, dianggap relatif kecil untuk dilakukan *brute force attack*. Namun proyek ini tidak terlalu berkembang karena tidak lama kemudian, pada bulan Agustus 2005 ditemukan *full collision* oleh para peneliti dari Cina.

Pada bulan Agustus 2004, diumumkan terjadinya *collision* untuk proses MD5 secara keseluruhan. Perlu diperhatikan bahwa *collision* yang terjadi sebelum ini hanya terjadi pada sebagian proses enkripsi MD5 saja, bukan secara keseluruhan. *Collision* ini diumumkan oleh peneliti dari Cina, Xiaoyun Wang, Dengguo Feng, Xuejia Lai dan Hongbo Yu. Proses ini hanya memerlukan waktu sekitar satu jam pada komputer IBM P690. Publikasi ini benar-benar menjadi perhatian pada analis kriptografi pada waktu itu.

Oktober 2004, Hawkes, Paddon, dan Rose mencoba memberikan penjelasan tambahan mengenai penelitian yang dilakukan Wang, dkk. Pada publikasinya Hawkes, dkk memberikan kondisi-kondisi yang harus dipenuhi oleh pesan agar dapat terjadi sebuah *collision*. Akan tetapi inti dari algoritma Wang, dkk tetap belum terpecahkan.

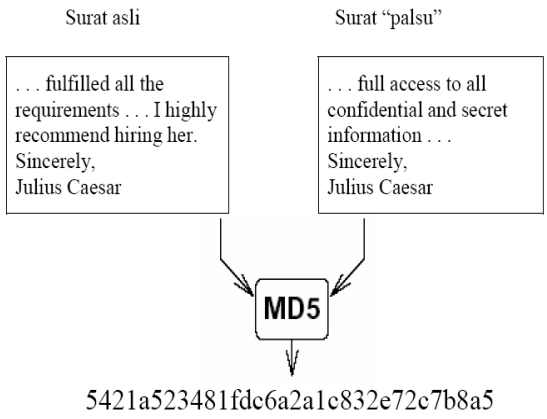
Vlastimil Klima menjelaskan sebuah algoritma yang lebih baik lagi untuk menemukan *collision* pada MD5, pada bulan Maret 2005. Dengan hanya menggunakan sebuah *notebook*, ia menemukan *full collision* pada MD5 dalam beberapa jam saja.

Tabel di bawah ini merupakan contoh kolisis pada MD5 hasil percobaan Xiaoyun Wang, Dengguo Feng, Xuejia Lai dan Hongbo Yu.

Tabel 7 – Tabel Kolisi MD5 hasil percobaan Xiaoyun Wang, Dengguo Feng, Xuejia Lai dan Hongbo Yu

X1	M	2dd31d1 c4eee6c5 69a3d69 5cf9af98 **87b5ca2f** ab7e4612 3e580440 897ffbb8 634ad55 2b3f409 8388e483 **5a417125** e8255108 9fc9cdf7 **f2bd1dd9** 5b3c3780
	N1	d11d0b96 9c7b41dc f497d8e4 d555655a c79a7335 cfdebf0 66f12930 8fb109d1 797f2775 eb5cd530 baade822 5c15cc79 ddcb74ed 6dd3c55f d80a9bb1 e3a7cc35
X2	M	2dd31d1 c4eee6c5 69a3d69 5cf9af98 **7b5ca2f** ab7e4612 3e580440 897ffbb8 634ad55 2b3f409 8388e483 **5a41f125** e8255108 9fc9cdf7 **72bd1dd9** 5b3c3780
	N1	d11d0b96 9c7b41dc f497d8e4 d555655a 479a7335 cfdebf0 66f12930 8fb109d1 797f2775 eb5cd530 baade822 5c154c79 ddcb74ed 6dd3c55f 580a9bb1 e3a7cc35
Nilai Hash		9603161f f41fc7ef 9f65ffbc a30f9dbf

Gambar di bawah ini merupakan contoh kolisi MD 5 oleh Stefan Lucks, Magnus Daum :



Gambar 8 Contoh kolisi MD5 oleh Stefan Lucks dan Magnus Daum

8. Percobaan Menggunakan Perangkat Lunak

Hasil Percobaan bermacam-macam input :

Kasus Test 1
Kasus test satu digunakan dengan mencoba input string "hello world" pada ketiga algoritma.

input : hello world
MD2: d9cce882ee690a5c1ce70beff3a78c77
MD4: aa010fbc1d14c795d86ef98c95479d17
MD5: 5eb63bbbe01eeed093cb22bb8f5acdc3

Kasus Test 2
Kasus test dua digunakan dengan mencoba input string "hello world" pada ketiga algoritma.

input : ini adalah contoh input untuk percobaan
MD2: 3c69fd0410e09b1c8f609864ea367d5a
MD4: 573b2f4010dd9a89b79bbc607a635b99
MD5: 6ee945fde7a99cb6da811d3ba26dcec0

Kasus Test 3
Kasus test 3 merupakan test untuk mencoba sebuah string kosong pada ketiga algoritma. Dapat dilihat pada hasilnya ketiga algoritma ini menghasilkan nilai *hash* yang berbeda untuk sebuah string kosong. Walaupun nilai inisialisasi variable penyangga untuk ketiga algoritma sama.

input :
MD2: 8350e5a3e24c153df2275c9f80692773
MD4: 31d6cfe0d16ae931b73c59d7e0c089c0
MD5: d41d8cd98f00b204e9800998ecf8427e

Kasus Test 4
Kasus test 4 menambahkan sebuah karakter pada kasus test 3. Dengan penambahan hanya satu karakter saja dihasilkan nilai *hash* yang berbeda dari hasil pada kasus test 3 untuk ketiga jenis algoritma *MD2*, *MD4*, *MD5*.

input : a
MD2: 32ec01ec4a6dac72c0ab96fb34c0b5d1
MD4: bde52cb31de33e46245e05fbdbd6fb24
MD5: 0cc175b9c0f1b6a831c399e269772661

Kasus Test 5
Kasus test 5 ini mencobakan sebuah string "test" pada ketiga algoritma.

input : test
MD2: dd34716876364a02d0195e2fb9ae2d1b
MD4: db346d691d7acc4dc2625db19f9e3f52
MD5: 098f6bcd4621d373cade4e832627b4f6

Kasus Test 6
Kasus test 6 mencobakan penambahan beberapa kata sebagai input pada kasus test 5.

input : test percobaan message digest
MD2: 3fbd4cd2153f346729f1693fe17e61b4
MD4: d60fc74aff967c533f41a4f77524c7cd
MD5: 2003501638c6fd9ec1c625ada868511f

9. Kesimpulan dan Saran

Ketiga algoritma di atas : *MD2*, *MD4*, dan *MD5* sebenarnya tidak begitu berbeda dalam segi kompleksitas dan teknik dalam proses menghasilkan *message digest*. Namun untuk kompleksitas waktu algoritma *MD4* yang memang didesain untuk meningkatkan performansi kecepatan memiliki hasil yang lebih baik. Namun dalam segi keamanan *MD4* lebih buruk dibandingkan algoritma lainnya.

Bentuk-bentuk *signature* yang menggunakan MD2 tidak begitu beresiko, tetapi MD2 tidak lagi direkomendasikan untuk aplikasi-aplikasi di masa mendatang, terutama aplikasi yang membutuhkan ketahanan terhadap kolisi (*collision-resistant*). Namun *MD2* masih merupakan fungsi satu arah.

MD4 sebaiknya tidak digunakan, apalagi setelah munculnya *MD5* yang memiliki tingkat keamanan yang lebih baik disbandingkan *MD4*. Namun, *MD5* pun memiliki kelemahan, baik *pseudo-collision* maupun kolisi telah banyak didemostrasikan oleh para ahli.

Bentuk-bentuk *signature* yang menggunakan *MD5* tidak beresiko dan *MD5* masih memiliki kemampuan untuk digunakan pada berbagai macam aplikasi di masa yang akan datang. Karena salah satu sifat seperti fungsi satu arah dan menghasilkan nilai output yang acak masih memungkinkan untuk *MD5*. Namun, sebaiknya *MD5* tidak digunakan untuk aplikasi yang sangat membutuhkan ketahanan terhadap kolisi (*collision resistant*).

Namun untuk memenuhi kebutuhan terhadap algoritma *hash* dalam berbagai aplikasi, algoritma *MD5* masih cukup dapat dipercaya. Tetapi masih banyak pilihan algoritma lain yang sama baiknya ataupun lebih baik seperti *SHA-1* dan *RIPEMD-160* yang merupakan alternatif yang cukup baik sebagai pilihan untuk fungsi *hash* dalam berbagai macam aplikasi.

Daftar Pustaka

[1] Bishop,David*, Introduction to Cryptography with Java Applet*, Jones and Bartlet Computer Science, 2003.

[2] Black, John. *A Study of the MD5 Attacks : Insight and Improvements*, University of Colorado.

[3] Kaliski, B, *The MD2 Message-Digest Algorithm*, RSA Laboratories, 1992.

[4] Munir, Rinaldi, *Diktat Kuliah IF5054 Kriptografi*, Sekolah Teknik Elektro dan Informatika Institut Teknologi Bandung, 2006.

[5] Rivest, Ron. *RFC 1320 – The MD4 Message-Digest Algorithm*, MIT Laboratory for Computer Science and RSA Data Security, Inc, 1992.

[6] Robshaw, M.J.B, *On Recent Result for MD2, MD4, and MD5*, 1996.

[7] Schneier, Bruce. (1996). Applied Cryptography 2nd. John Wiley & Sons.

[8] X.Wang, D. Feng, X. Lai, H. Yu, *Collision for Hash Function MD4, MD5, HAVAL-128 and RIPEMD*, rump session, CRYPTO 2004, *Cryptology ePrint Archive*, Report 2004/199.