SERANGAN KOLISI MD5 PADA DUPLIKASI EKSTRAKSI *FILE* PAKET

Masykur Marhendra S.N. – NIM : 13504063

Program Studi Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
E-mail : if14063@students.if.itb.ac.id

Abstrak

Makalah ini membahas tentang peranan algoritma MD5 pada proses ekstraksi *file* paket dan pembuatan *file* paket. Dalam pembuatan paket, kita masukkan *magic number* pada header paket dan diikuti oleh data dari paket, yaitu data-1 dan data-2. Pada akhir pembuatan paket akan kita dapatkan nilai MD5 yang sama antara dua paket, yaitu data-1.pak dan data-2.pak. Hal ini karena kolisi MD5 yang terjadi pada *magic number*. Berikutnya, ketika kita mengekstraksi kembali arsip paket tersebut, akan kita dapatkan arsip hasil ekstraksi yang berbeda antara data-1.pak dan data-2.pak. Penentuan arsip mana yang akan diekstraksi ditentukan oleh nilai *bitmask* yang digunakan untuk ekstraksi.

Makalah ini juga akan membahas tentang algoritma MD5 dan sedikit tentang kolisi MD5. Penjelasan ini diharapkan dapat memberikan gambaran dan wawasan dalam memahami serangan MD5 pada ekstraksi *file* paket ini. Begitu algoritma untuk menemukan kolisi MD5 dipublikasikan, serangan MD5 tersebut dapat dirancang lebih efektif. Diantaranya ketika menciptakan arsip paket, kita samarkan data yang ada di dalam paket tersebut dan kita paketkan bersama dengan program ekstraksinya.

***Kata kunci**: collision byte, fungsi hash, kolisi, magic number, message digest, MD5, word.*

1. Pendahuluan

Perkembangan dunia digital saat ini membuat lalu lintas pengiriman data elektronik semakin ramai. Hampir setiap orang melakukan transaksi data setiap harinya. Data yang dipertukarkan pun juga bervariasi baik dari jenisnya maupun tingkat autentikasinya. Mulai dari data pribadi,data organisasi sampai dengan data perusahaan yang diperjualbelikan. Hal inilah yang menuntut adanya sistem autentikasi data sehingga data tidak sampai terduplikasi oleh pihak ketiga dan merugikan kedua belah pihak, yaitu pihak penerima maupun pengirim. Sampai saat ini telah banyak ditemukan teknik-teknik dalam melakukan autentikasi data, baik teknik klasik maupun modern.

Fungsi *hash* adalah salah satu algoritma yang digunakan untuk melakukan autentikasi data . Fungsi *hash* ini mendasari beberapa algoritma autentikasi pesan lainnya seperti MAC, MD5, dsb. Pada dasarnya, fungsi *hash* mengkompresi pesan dengan panjang yang berbeda-beda menjadi pesan acak yang mempunyai panjang yang terdefinisi (umumnya berukuran jauh lebih kecil daripada ukuran pesan semula).

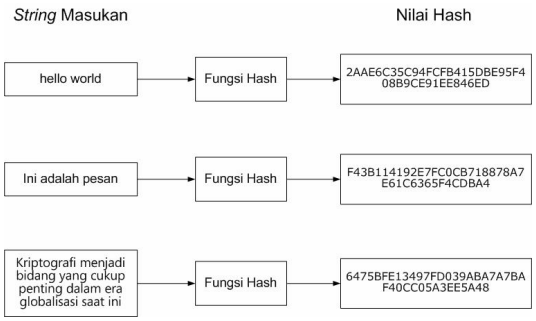
Fungsi *hash* dapat menerima masukan *string* apa saja. Jika *string* menyatakan pesan (*message*), maka sembarang pesan *M* berukuran bebas dikompresi oleh fungsi *hash H* melalui persamaan :

$$h = H(M) \quad (1)$$

Keluaran fungsi *hash* disebut juga **nilai *hash*** (*hash-value*) atau **pesan-ringkas** (*message digest*). Pada persamaan (1), *h* adalah nilai *hash* atau *message digest* dari fungsi *H* untuk masukan *M*. Dengan kata lain, fungsi *hash* mengkompresi sembarang pesan yang berukuran berapa saja menjadi *message digest* yang ukurannya selalu tetap. Gambar 1 berikut memperlihatkan contoh tiga buah pesan dengan panjang yang berbeda-beda selalu di-*hash* menghasilkan pesan ringkas yang panjangnya tetap (dalam contoh ini pesan ringkas dinyatakan dalam kode heksadesimal yang panjangnya 128 bit, satu karakter heksadesimal = 4 bit).

Aplikasi fungsi *hash* misalnya untuk memverifikasi kesamaan salinan suatu arsip dengan arsip aslinya yang tersimpan di dalam sebuah basis data terpusat. Melakukan pengiriman *message digest* akan lebih mangkus daripada melakukan pengiriman salinan arsip

tersebut secara keseluruhan ke komputer pusat (yang membutuhkan waktu transmisi lama dan ongkos mahal). Jika *message digest* salinan arsip sama dengan *message digest* arsip asli, berarti salinan arsip tersebut sama dengan arsip di dalam basis data.



Gambar 1 Contoh *hash*ing beberapa buah pesan dengan panjang berbeda-beda

2. Fungsi *Hash* Satu Arah

Fungsi *hash* satu arah adalah fungsi *hash* yang bekerja dalam satu arah : pesan yang sudah dipesan menjadi *message digest* tidak dapat dikembalikan lagi menjadi pesan semula. Dua pesan yang berbeda akan selalu menghasilkan nilai *hash* yang berbeda pula. Sifat-sifat fungsi *hash* satu arah adalah sebagai berikut [SCH96]:

1. Fungsi H dapat diterapkan pada blok data berukuran berapa saja.
2. *H* menghasilkan nilai (*h*) dengan panjang tetap (*fixed-length-output*).
3. *H(x)* mudah dihitung untuk setiap nilai *x* yang diberikan.
4. Untuk setiap h yang diberikan, tidak mungkin menemukan x sedemikian hingga H(x) = h. Itulah sebabnya fungsi H dikatakan

Keenam sifat diatas penting sebab sebuah fungsi *hash* seharusnya berlaku seperti fungsi acak. Namun, ada beberapa kasus dimana dapat terjadi suatu kolisi (*collision*), yang akan dibahas selanjutnya, yaitu ketika terdapat dua pesan yang berbeda yang mempunyai pesan ringkas yang sama.

Fungsi *hash* bekerja secara iteratif. Masukan fungsi *hash* adalah blok pesan (*M*) dan keluaran dari *hashing* blok pesan sebelumnya.

$$h_i = H(M_i, h_{i-1}) \qquad (2)$$

Secara tepat, sebuah fungsi *hash* h memetakan bit-strings dengan panjang bervariasi menjadi bit-strings dengan panjang n bits. Untuk sebuah *domain D* dan *range R* dengan $h : D \to R$ dan $|D| > |R|$. Tentu saja, *h* terbatas terhadap domain input t-bit ($t > n$), jika h sebuah random dimana semua keluaran dapat diterima, maka kira-kira 2^{t-n} masukan dapat memetakan ke masing-masing keluaran, dan dua masukan acak dapat menghasilkan keluaran yang sama dengan probabilitas 2^{-n} (dengan *t* adalah variabel bebas).

Skema fungsi *hash* ditunjukkan pada gambar 2.1. Fungsi *hash* adalah publik (tidak dirahasiakan), dan keamanannya terletak pada sifat satu arahnya itu. Ada beberapa fungsi *hash* satu arah yang telah diciptakan, antara lain adala MD4, MD5, *Secure Hash Algorithm* (SHA), RIPEMD, WHIRLPOOL, dan lain lain. Dan MD5 adalah algoritma yang secara luas dipakai dalam melakukan otentikasi data atau pesan dan digunakan sebagai *message digest* dalam pembuatan tanda tangan digital.

Gambar 2 Fungsi *hash* satu arah

3. MD5 sebagai Fungsi *Hash* Satu Arah

3.1 Prolog MD5

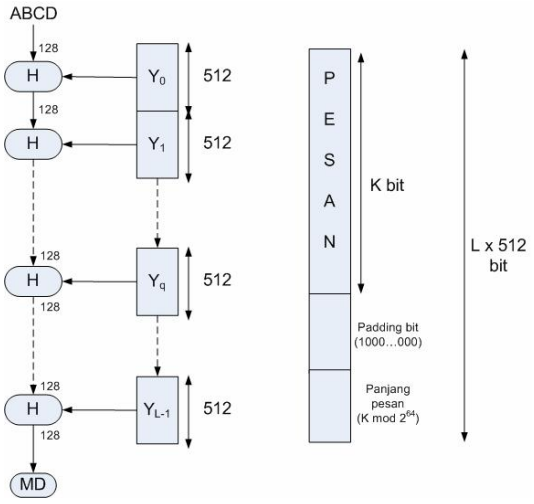
MD5 didesain oleh Profesor Ronald Rivest, pencipta algoritma RSA, pada tahun 1991 berupa fungsi *hash* satu arah. MD5 merupakan fungsi *hash* pengganti predesornya, MD4, karena dianggap tidak aman lagi setelah adanya serangan analitik yang melemahkan algoritma tersebut.

Algoritma MD5 secara umum lebih lambat daripada MD4, tetapi lebih 'konservatif' dalam design. MD5 didesign karena MD4 dirasa mungkin diadopsi untuk pemakaian yang cepat daripada merapikan sisi kritikal; karena MD4 didesign untuk beroperasi cepat, maka MD4 memberikan peluang lebih besar kepada kriptanalisis untuk melakukan serangan-serangan. Lain halnya dengan MD5, fungsi *hash* ini lebih memberikan perhatian lebih ke tingkat

keamanan yang berganda dan sedikit menyerah terhadap kecepatan operasi.

3.2 Algoritma MD5

Algoritma MD5 menerima masukan berupa pesan dengan ukuran sembarang dan menghasilkan *message digest* yang panjangnya 128 bit. Gambaran pembuatan *message digest* dengan algoritma MD5 diperlihatkan pada gambar 3



Gambar 3 Pembuatan *message digest* dengan algoritma MD5

Langkah-langkah pembuatan *message digest* secara garis besar adalah sebagai berikut :

1. Penambahan bit-bit pengganjal (*padding bits*)
2. Penambahan nilai panjang pesan semula
3. Inisiasi penyangga (*buffer*) MD. [RIN06]

Pada makalah ini sebuah *word* adalah sebuah kuantitas 32-*bit* dan sebuah *byte* adalah sebuah kuantitas 8-*bit*. *Sequence of bits* dapat direpresen tasikan sebagai *sequence of bytes*, dimana setiap 8-bit blok berurutan direpresentasi kan sebagai sebuah *byte* dengan *high-order* (*most significant*) *bit* dari setiap *byte*. Dan sebaliknya, sequence of *byte*s dapat juga di representasikan sebagai 32-*bit word* sekuens, dimana setiap 4-*bytes* blok berurutan direpresentasikan sebagai *word* dalam *low-order* (*least significant*) *byte* seperti telah didefinisikan sebelumnya.

Pandang simbol "+" sebagai penambahan *word*s (yaitu, penambahan dengan modulo 2^{32}).

Pandang $X <<< s$ sebagai nilai 32-bit dari hasil *circular shift X* ke kiri sebanyak *s-bit*. Pandang $not(X)$ sebagai *bit-wise complement* dari X, dan pandang $X \vee Y$ sebagai *bit-wise OR* dari X dan Y . Pandang juga $X \oplus Y$ sebagai *bit-wise XOR* dari *X* dan *Y*, dan pandang $X \wedge Y$ sebagai *bit-wise AND* dari *X* dan *Y*.

Kita mulai dengan mengasumsikan bahwa kita mempunyai pesan *b-bit* sebagai masukan, dan sehingga kita ingin mendapatkan *message digest*-nya. Nilai *b* adalah variabel integer non negatif ; *b* bisa nol. Pandang *bits* dari pesan seperti yang tertulis dibawah :

$$m_0, m_1, m_2 \cdots m_{(b-1)}$$

Berikut adalah definisi dari setiap langkah pembuatan *message digest* MD5.

3.2.1 Penambahan bit pengganjal

Pesan ditambah dengan sejumlah bit pengganjal sedemikian sehingga panjang bit dari pesan kongruen dengan 448 modulo 512 ($l \equiv 448 \bmod 512$). Hal ini menjelaskan bahwa panjang pesan setelah penambahan bit-bit pengganjal adalah 64-*bit* kurang dari 512. Nilai 512 muncul karena pemrosesan MD5 per 512-*bit* tiap bloknya. Pesan dengan ukuran 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 berupa sebuah bit 1 dan kemudian diikuti dengan bit 0 untuk sisanya.

3.2.2 Penambahan nilai panjang pesan semula

Pesan yang telah diberi bit-bit pengganjal sebelumnya kemudian ditambah (*append*) lagi dengan representasi 64-*bit b* pesan. Jika nilai *b* lebih besar dari 2^{64} maka yang diambil hanya *lower-order* 64-*bit* dari *b*, atau dengan kata lain panjang *b* dalam modulo 2^{64} .

Setelah ditambah dengan 64-*bit*, panjang pesan akan menjadi kelipatan 512-*bit* (16-*word*). Atau dapat dikatakan bahwa pesan ini mempunyai panjang yang merupakan kelipatan dari 16-*words* (32-*bit*). Pandang $M[0...(N-1)]$ sebagai *words* dari pesan hasil penambahan bit dan panjang pesan itu sendiri, dimana N adalah kelipatan 16.

3.2.3 Inisiasi penyangga MD

MD5 membutuhkan empat buah *word* penyangga (*buffer*) yang masing-masing panjangnya 32-bit. Total panjang penyangga adalah $4 \cdot 32 = 128$ bit. Keempat penyangga ini menampung hasil antara dan hasil akhir. Pandang A, B, C, D sebagai empat *word* buffer penyangga dengan setiap nilainya dalam heksadesimal adalah sebagai berikut :

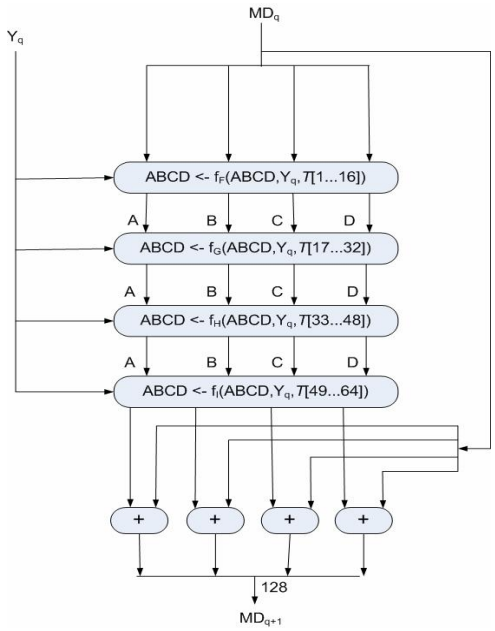
```
word A: 01 23 45 67
word B: 89 ab cd ef
word C: fe dc ba 98
word D: 76 54 32 10
```

Namun, pada beberapa versi lainnya inisiasi *word* penyangga ini dapat berbeda seperti dibawah ini [RIN06] :

```
word A: 67 54 32 10
word B: ef cd ab 89
word C: 98 ba dc fe
word D: 10 32 54 67
```

3.2.4 Pengolahan pesan dalam blok 16-*word*

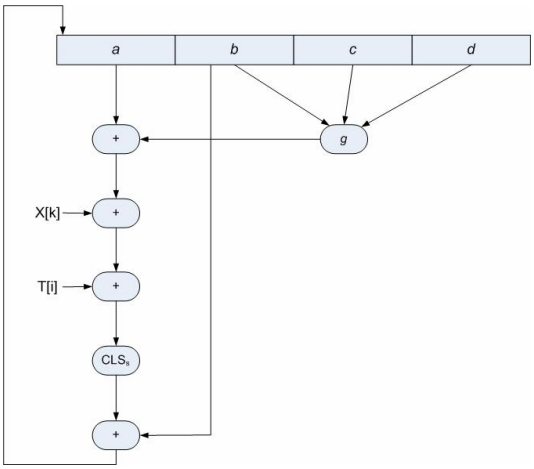
Pesan dibagi menjadi *L* buah blok yang masing-masing panjangnya 16-*word* (Y_0 sampai Y_{L-1}). Setiap blok 16-*word* diproses bersama dengan penyangga *MD* menjadi keluaran 8-*word*, dan ini disebut proses H_{MD5}. Gambaran proses H_{MD5} dapat diperlihatkan pada Gambar 4 dibawah ini:



Gambar 4 Pengolahan blok 16-*words*

Proses H_{MD5} terdiri dari 4 buah putaran dan masing-masing putaran melakukan operasi dasar *MD5* sebanyak 16 kali dan setiap operasi dasar memakai sebuah elemen *T*. Jadi setiap putaran memakai 16 elemen Tabel *T*. Pada Gambar 3.2, Y_q menyatakan blok 16-*word* 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-bitdari proses H_{MD5} ke-*q*. Pada awal proses, MD_q berisi nilai inisiasi penyangga *MD*.

Fungsi-fungsi f_F, f_G, f_H, f_I masing-masing berisi 16 kali operasi dasar terhadap masukan, setiap operasi dasar menggunakan elemen Tabel *T*. Operasi dasar MD5 diperlihatkan pada Gambar 5 dibawah ini,



Gambar 5 Operasi dasar MD5

Operasi tersebut dapat dituliskan menjadi sebuah persamaan dibawah ini :

$$a \leftarrow b + CLS_s(a + g(b,c,d) + X[k] + T[i]) \quad (3)$$

yang dalam hal ini :

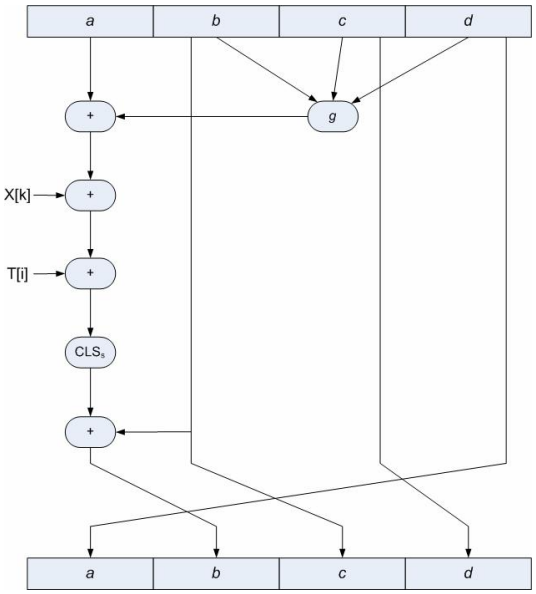
a, b, c, d = empat buah *word* penyangga
g = salah satu fungsi *F*, *G*, *H*, *I*
CLS_s = *circular left shift* sebanyak s-bit (<<< *s*)
$X[k]$ = kelompok 32-bit ke-k dari blok 16-*word message* ke-*q*. Nilai $k = 0$ sampai 15.
$T[i]$ = elemen Tabel *T* ke-*i* (32-*bit*).

Karena ada 16 kali operasi dasar, maka setiap kali selesai satu operasi dasar, penyangga-

penyangga itu digeser ke kanan secara sirkuler dengan cara pertukaran yang dapat didefinisikan sebagai berikut [RIN06] :

$$temp \leftarrow d$$
$$d \leftarrow c$$
$$c \leftarrow b$$
$$b \leftarrow a$$
$$a \leftarrow temp$$

Pergeseran tersebut dapat digambarkan pada Gambar 6, 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, f_I adalah fungsi untuk memanipulasi masukan a, b, c, dan d dengan ukuran 32-*bit word* dan menghasilkan keluaran 32-*bit word*. Setiap definisi fungsi dapat dilihat pada tabel dibawah ini :

Tabel 1 Fungsi-fungsi dasar MD5

Nama	Notasi	$g(b,c,d)$
f_F	$F(b,c,d)$	$(b \wedge c) \vee (\sim b \wedge d)$
f_G	$G(b,c,d)$	$(b \wedge d) \vee (c \wedge \sim d)$
f_H	$H(b,c,d)$	$b \oplus c \oplus d$
f_I	$I(b,c,d)$	$c \oplus (b \vee \sim d)$

Tabel $T[i]$ dapat dilihat pada Tabel 2. Tabel ini disusun oleh fungsi $t(i) = 2^{32} \cdot abs(\sin(i))$ yang dalam hal ini *i* dalam radian dan dalam notasi heksa desimal.

Tabel 2 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

Pada persamaan (3) dapat dilihat bahwa masing-masing fungsi f_F, f_G, f_H, f_I melakukan 16 kali operasi dasar. Misalkan notasi

```
[abcd k s i]
```

menyatakan operasi

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

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 3

Tabel 4 Rincian operasi pada fungsi F(b,c,d)

[ABCD 0 7 1]	[DABC 1 12 2]
[CDAB 2 17 3]	[BCDA 3 22 4]

[ABCD	4	7	5]	[DABC	5	12	6]
[CDAB	6	17	7]	[BCDA	7	22	8]
[ABCD	8	7	9]	[DABC	9	12	10]
[CDAB	10	17	11]	[BCDA	11	22	12]
[ABCD	12	7	13]	[DABC	13	12	14]
[CDAB	14	17	15]	[BCDA	15	22	16]

Putaran 2

16 kali operasi dasar dengan

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

diberikan pada tabel 5

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

[ABCD	1	5	17]	[DABC	6	9	18]
[CDAB	11	14	19]	[BCDA	0	20	20]
[ABCD	5	5	21]	[DABC	10	9	22]
[CDAB	15	14	23]	[BCDA	4	20	24]
[ABCD	9	5	25]	[DABC	14	9	26]
[CDAB	3	14	27]	[BCDA	8	20	28]
[ABCD	13	5	29]	[DABC	2	9	30]
[CDAB	7	14	31]	[BCDA	12	20	32]

Putaran 3

16 kali operasi dasar dengan

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

diberikan pada tabel 6

Tabel 6 Rincian operasi pada fungsi *H(b,c,d)*

[ABCD	5	4	33]	[DABC	8	11	34]
[CDAB	11	16	35]	[BCDA	14	23	36]
[ABCD	1	4	37]	[DABC	4	11	38]
[CDAB	7	16	39]	[BCDA	10	23	40]
[ABCD	13	4	41]	[DABC	0	11	42]
[CDAB	3	16	43]	[BCDA	6	23	44]
[ABCD	9	4	45]	[DABC	12	11	46]
[CDAB	15	16	47]	[BCDA	2	23	48]

Putaran 4

16 kali operasi dasar dengan

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

diberikan pada tabel 7

Tabel 7 Rincian operasi pada fungsi *I(b,c,d)*

[ABCD	0	6	49]	[DABC	7	10	50]
[CDAB	12	6	53]	[DABC	3	10	54]
[CDAB	10	15	55]	[BCDA	1	21	56]
[ABCD	8	6	57]	[DABC	15	10	58]
[CDAB	6	15	59]	[BCDA	13	21	60]
[ABCD	4	6	61]	[DABC	11	10	62]
[CDAB	2	15	63]	[BCDA	9	21	64]

Setelah putaran keempat selesai, kemudian nilai *a*, *b*, *c*, dan *d* ditambahkan ke *A*, *B*, *C*, *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*, *D*.

Dari uraian diatas, secara umum fungsi *hash* MD5 dapat dituliskan sebagai persamaan matematika berikut :

$$MD_0 = IV$$

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

$$MD = MD_{L-1}$$

yang dalam hal ini :

IV = vektor inisialisasi dari *word* 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*

Berikut adalah *pseudo-code* MD5 dalam notasi mirip bahasa C :

```
//Catatan: Semua peubah adalah
32 bits tidak bertanda dan
penjumlahan dalam bentuk modulo
2^32

//Definisikan r sebagai berikut
var int[64] r, T

// r[0..15]
r[ 0..15] := {7, 12, 17, 22,
7, 12, 17, 22,  7, 12, 17, 22,
7, 12, 17, 22}

// r[16..31]
r[16..31] := {5,  9, 14, 20,
5,  9, 14, 20,  5,  9, 14, 20,
5,  9, 14, 20}
```

```
// r[32..47]
r[32..47] := {4, 11, 16, 23,
4, 11, 16, 23,  4, 11, 16, 23,
4, 11, 16, 23}

// r[48..63]
r[48..63] := {6, 10, 15, 21,
6, 10, 15, 21,  6, 10, 15, 21,
6, 10, 15, 21}

//Gunakan integer biner dari
fungsi sinus  sebagai konstanta
T:
for i from 0 to 63
    T[i] := floor(abs(sin(i +
1)) × 2^32)

//Inisialisasi variabel dimana
h0 = A; h1= B; h2 = C; h3 = D
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476

//Pemrosesan awal:
append bit "1" ke pesan
append sejumlah bit "0" until
panjang pesan dalam bit ≡ 448
(mod 512)
append panjang bit pesan as 64-
bit little-endian integer to
pesan

//Proses pesan dalam blok-blok
berukuran 512-bit
for each 512-bit blok pesan
    bagi pesan menjadi 16 buah
sub-blok 32-bit little-endian
words w(i), 0 ≤ i ≤ 15

//Inisialisasi peubah penyangga
var int a := h0
var int b := h1
var int c := h2
var int d := h3

//Kalang utama:
for i from 0 to 63
   if 0 ≤ i ≤ 15 then
      f := (b and c) or
           ((not b) and d)
      g := i
   else if 16 ≤ i ≤ 31
      f := (d and b) or
           ((not d) and c)
      g := (5×i + 1) mod 16
   else if 32 ≤ i ≤ 47
      f := b xor c xor d
      g := (3×i + 5) mod 16
   else if 48 ≤ i ≤ 63
      f := c xor (b or (not d))
      g := (7×i) mod 16

// Geser penyangga ke kanan
   secara sirkuler
temp := d
d := c
c := b
b := ((a + f + T[i] + w(g))
      leftrotate r[i]) + b
a := temp

//Jumlahkan blok blok ini
h0 := h0 + a
h1 := h1 + b
h2 := h2 + c
h3 := h3 + d

//Pesan ringkas sebagai
  representasi little-endian
var int digest :=
      h0 append h1
         append h2
         append h3
```

Program 1 *Pseudo-code* algoritma MD5

3.3 Kriptanalisis terhadap algoritma MD5

MD5 telah menjadi sorotan publik kriptografi sejak pertama kali dipublikasikan. Pada tahun 2004, hampir semua riset serangan terhadap MD5 hanya dapat menunjukkan kelemahan – kelemahan kecil pada desain algoritmanya. Meskipun ada dua serangan yang dapat menunjukkan adanya permasalahan serius pada desain.

Indikasi awal yang memperlihatkan bahwa terdapat kelalaian dalam desain MD5 adalah sebuah makalah yang dipublikasikan oleh Den Boer dan Booselaer. Mereka mendemonstrasikan bahwa dengan beberapa kondisi masukan yang berbeda, terdapat peluang untuk mendapatkan nilai MD5 yang sama.

Indikasi kedua terjadi pada tahun 1996 ketika Dobbertin mendemonstrasikan bahwa nilai MD5 yang sama dapat diperoleh apabila vektor inisialisasi dapat ditentukan [DOB96]. Namun, hasil riset dari Dobbertin ini tidak memberikan kekhawatiran pada keamanan krena pada

praktiknya vektor inisial MD5 telah ditentukan sesuai konvensi dengan nilai yang sama (IV_0). Meski demikian, Dobbertin telah berhasil memperlihatkan bahwa kolisi MD5 tidak dapat dihindarkan.

Pada tanggal 1 Maret 2005, Arjen Lenstra, Xiaoyun Wang, dan Benne de Weger mendemonstrasikan pembentukan dua buah sertifikat X.509 dengan kunci publik yang berbeda tetapi mempunyai nilai *hash* yang sama. Sebelumnya, mereka mendemonstrasikan kolisi MD5 dengan menggunakan vektor inisialisasi standar (IV_0). Riset ini memperlihatkan bahwa memungkinkan untuk dapat menciptakan masukan 512-bit dan memodifkasi bit-bit tertentu untuk menciptakan dua pesan yang berbeda dengan nilai *hash* yang sama. Berikut adalah salah satu contoh dua pesan yang memiliki nilai *hash* yang sama [WANG04] :

Tabel 8 Dua blok pesan yang saling berkolisi

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

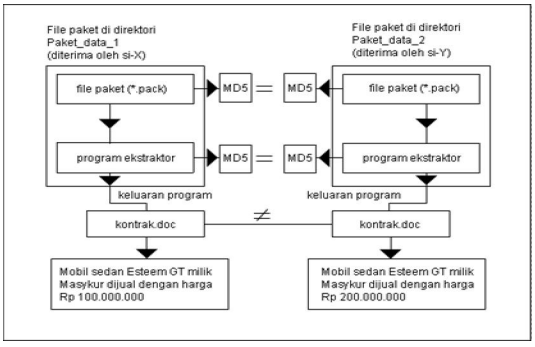
3.4 Kolisi MD5 pada Pembuatan *File* Paket

3.4.1. Gambaran umum

File paket merupakan suatu *file* yang mengemas dua atau lebih *file* menjadi satu sehingga mudah untuk memindahkan *file* tersebut. *File* paket biasanya digunakan pada data atau *file* yang saling dependen. Hasil ekstraksi *file* satu bergantung pada hasil ekstraksi *file* sebelumnya. Dan *file* tersebut hanya bisa diekstraksi apabila *file* sebelumnya telah berhasil diekstraksi.

Pada makalah ini, *file* paket juga dibuat untuk mengemas data, namun hanya terbatas pada dua data saja. Tujuan utama dari pembuatan *file* paket pada makalah ini adalah untuk menunjukkan redundansi data pada hasil ekstraksi, sehingga apabila *file* tersebut didistribusikan dan diekstraksi, akan didapatkan dua *file* yang benar-benar berbeda.

Misalkan terdapat sebuah *file A* yang akan dipaketkan dan seseorang menginginkan agar ketika *file* paket tesebut didistribusikan dan diekstraksi adalah *file* lain, maka dia cukup menambahkan *file* B pada paket tesebut dan mendistribusikan *file* paket yang mempunyai hasil ekstraksi *file B*. Si penerima ketika mengecek keaslian paket dengan nilai *hash* , akan selalu mendapatkan bahwa *file* paket tesebut adalah *file* yang benar, padahal sebenarnya tidak. Hal ini karena dua *file* paket yang dibuat mempunyai nilai *hash* yang sama.



Gambar 7 Skema pemaketan dan pengekstraksian dua data *file* paket

Pada Gambar 7 diatas, terdapat dua *file* paket yang sama, yaitu yang memuat *file* kontrak.doc. Namun dari masing-masing *file* paket, isi dari

kontrak.doc berbeda satu sama lain. Ketika ekstraksi berjalan, program akan dapat menentukan *file* mana yang akan diekstraksi sehingga dihasilkan *file* kontrak.doc yang isinya berbeda. Kedua *file* paket tersebut sebelumnya lolos dari pengujian otentikasi dengan algoritma MD5, karena keduanya mempunyai nilai MD5 yang sama.

3.4.2. Lingkungan implementasi

Program aplikasi algoritma transposisi ini dikembangkan pada lingkungan sebagai berikut:

- **Prosesor** Intel(R) Pentium(R) 4 2,66 GHz Core Speed 2660.8 MHz Bus Speed 532.2 MHz
- **Data Cache**
  - L1 8-ways 16 KB
  - L2 8-ways 1024 KB
- **Memory** Visipro DDR-SDRAM 512 MB 166.3 MHz
- **VGA** Nvidia Ge-Force FX5200 128 MB
- **OS** Windows XP SP2 with Framework 2.0
- **Kakas pemrograman** Microsoft Visual Studio .NET berbasis C# language

3.4.3. Algoritma utama

3.4.3.1 Magic Number Collision

Magic number collision adalah sepasang *bytes* yang dapat menghasilkan nilai MD5 yang sama. *Magic number collision* didapatkan dengan menggunakan rumus yang didefinisikan X. Wang dkk. X. Wang mengatakan bahwa mereka mampu mendapatkan banyak kolisi yang tersusun atas dua pesan 1024-bit dengan vektor inisial asli MD5 IV_0 :
(dimana pada posisi 4,11,dan 14 tidak nol)

$$IV_0' : A_0' = 0x12AC2375,\ B_0' = 0x3B341042,$$
$$C_0' = 0x5F62B97C,\ D_0' = 0x4BA763ED$$

$$M' = M + \Delta C_1, \Delta C_1 = (0,0,0,0,\ 2^{31},\ldots,\ 2^{5},\ldots,\ 2^{31},0)$$

$$N_i' = + N_i + \Delta C_2, \Delta C_2 = (0,0,0,0,\ 2^{31},\ldots,-2^{5},\ldots,\ 2^{31},0)$$

sehingga :

$$MD5(M, N) = MD5(M1, N')$$

Pada makalah [10] lebih jelas dibahas tentang beberapa kondisi untuk menghasilkan pesan yang mempunyai dua nilai MD5 yang sama. Sedangkan pada [5] telah tersedia pseudo code generator kolisi MD5 (sengaja tidak dicantumkan karena terlalu panjang).

Berdasarkan generator tersebut, yang memanfaatkan rumus X. Wang, maka saya mendapatkan salah satu *magic number collision* sebagai berikut :

```
byte[] collisionFirstPart =
new byte[128]
{
 0xd1 , 0x31 , 0xdd , 0x02 ,
 0xc5 , 0xe6 , 0xee , 0xc4 ,
0x69 , 0x3d , 0x9a , 0x06 ,
0x98 , 0xaf , 0xf9 , 0x5c ,
0x2f , 0xca , 0xb5 , 0x87 ,
0x12 , 0x46 , 0x7e , 0xab ,
0x40 , 0x04 , 0x58 , 0x3e ,
0xb8 , 0xfb , 0x7f , 0x89 ,
0x55 , 0xad , 0x34 , 0x06 ,
0x09 , 0xf4 , 0xb3 , 0x02 ,
0x83 , 0xe4 , 0x88 , 0x83 ,
0x25 , 0x71 , 0x41 , 0x5a ,
0x08 , 0x51 , 0x25 , 0xe8 ,
0xf7 , 0xcd , 0xc9 , 0x9f ,
0xd9 , 0x1d , 0xbd , 0xf2 ,
0x80 , 0x37 , 0x3c , 0x5b ,
0xd8 , 0x82 , 0x3e , 0x31 ,
0x56 , 0x34 , 0x8f , 0x5b ,
0xae , 0x6d , 0xac , 0xd4 ,
0x36 , 0xc9 , 0x19 , 0xc6 ,
0xdd , 0x53 , 0xe2 , 0xb4 ,
0x87 , 0xda , 0x03 , 0xfd ,
0x02 , 0x39 , 0x63 , 0x06 ,
0xd2 , 0x48 , 0xcd , 0xa0 ,
0xe9 , 0x9f , 0x33 , 0x42 ,
0x0f , 0x57 , 0x7e , 0xe8 ,
0xce , 0x54 , 0xb6 , 0x70 ,
0x80 , 0xa8 , 0x0d , 0x1e ,
0xc6 , 0x98 , 0x21 , 0xbc ,
0xb6 , 0xa8 , 0x83 , 0x93 ,
0x96 , 0xf9 , 0x65 , 0x2b ,
0x6f , 0xf7 , 0x2a , 0x70
};
```

Program 2 *Colliding bytes* blok pertama

dan pasangan magic number lainnya adalah

```
byte[] collisionSecondPart =
new byte[128]
{
0xd1 , 0x31 , 0xdd , 0x02 ,
0xc5 , 0xe6 , 0xee , 0xc4 ,
0x69 , 0x3d , 0x9a , 0x06 ,
0x98 , 0xaf , 0xf9 , 0x5c ,
0x2f , 0xca , 0xb5 , 0x07 ,
0x12 , 0x46 , 0x7e , 0xab ,
0x40 , 0x04 , 0x58 , 0x3e ,
```

```
0xb8 , 0xfb , 0x7f , 0x89 ,
0x55 , 0xad , 0x34 , 0x06 ,
0x09 , 0xf4 , 0xb3 , 0x02 ,
0x83 , 0xe4 , 0x88 , 0x83 ,
0x25 , 0xf1 , 0x41 , 0x5a ,
0x08 , 0x51 , 0x25 , 0xe8 ,
0xf7 , 0xcd , 0xc9 , 0x9f ,
0xd9 , 0x1d , 0xbd , 0x72 ,
0x80 , 0x37 , 0x3c , 0x5b ,
0xd8 , 0x82 , 0x3e , 0x31 ,
0x56 , 0x34 , 0x8f , 0x5b ,
0xae , 0x6d , 0xac , 0xd4 ,
0x36 , 0xc9 , 0x19 , 0xc6 ,
0xdd , 0x53 , 0xe2 , 0x34 ,
0x87 , 0xda , 0x03 , 0xfd ,
0x02 , 0x39 , 0x63 , 0x06 ,
0xd2 , 0x48 , 0xcd , 0xa0 ,
0xe9 , 0x9f , 0x33 , 0x42 ,
0x0f , 0x57 , 0x7e , 0xe8 ,
0xce , 0x54 , 0xb6 , 0x70 ,
0x80 , 0x28 , 0x0d , 0x1e ,
0xc6 , 0x98 , 0x21 , 0xbc ,
0xb6 , 0xa8 , 0x83 , 0x93 ,
0x96 , 0xf9 , 0x65 , 0xab ,
0x6f , 0xf7 , 0x2a , 0x70
};
```

Program 3 *Colliding bytes* blok kedua

dimana nilai-nilai *byte* yang digarisbawahi adalah *byte-byte* yang saling berkolisi. Selanjutnya pada proses, nilai dari magic number ini akan diletakkan pada header *file* paket untuk dapat memberikan nilai MD5 yang sama serta sebagai penanda *file* mana yang akan diekstraksi.

3.4.3.2 Struktur Data *File* Paket

Secara sederhana *file* paket kustom ini terdiri atas header , nama *file* yang akan menjadi nama *file* hasil ekstraksi, panjang masing-masing *file* yang dipaketkan, dan data pada *file*-1 dan *file*-2 yang dipaketkan. Rincian dari setiap bagian dapat dilihat pada tabel dibawah ini :

Tabel 9 Struktur Data *File* Paket

Ukuran (*bytes*)	Data yang disimpan
128	blok *collidiing bytes*
1	panjang nama *file* -fnamelen
fnamelen	nama *file* yang akan diekstraksi
4	32-bit integer ukuran *file* pertama-*filesize1*
4	32-bit integer ukuran *file* kedua-*filesize2*
*file*size1	data dari *file*-1 yang ter-obfuscated
*file*size2	data dari *file*-2 yang terobfuscated

Blok *colliding bytes* antara *file* paket satu dengan lainnya berbeda. Sedangkan data sisanya selalu sama.

3.4.3.3 Pembuatan *File* Paket

Pada awal proses pembuatan *file* paket, program akan meminta masukan berupa dua *file* yang akan dipaketkan. Dua *file* ini bisa bertipe apapun. Di dalam program ini juga ditampikan nilai *hash* MD5 dari masing-masing *file* sehingga dapat mengetahui bahwa memang sebelumnya kedua *file* tersebut benar-benar mempunyai nilai *hash* yang berbeda. Kemudian, setelah itu *file* paket dibuat, yaitu dengan meletakkan blok *colliding byte*s pertama ke *file* paket-1 dan blok *colliding byte*s kedua ke *file* paket-2. Setelah itu diikuti penulisan data seperti yang telah didefinisikan di sub bab 3.4.3.2

Secara umum algoritma dari proses pembuatan *file* paket adalah sebagai berikut (dalam notasi sedikit bahasa C#):

```
function GetFileSize(
ref inFile: BinaryReader)→
uint32_t
{Mengembalikan ukuran dari file
yang sedang dibaca oleh
BinaryReader
}

Deklarasi
fsize : uint32_t

Algoritma
inFile.BaseStream.Seek(0,
SeekOrigin.End);

fsize := (uint32_t)
inFile.BaseStream.Position;

inFile.BaseStream.Seek(0,
SeekOrigin.Begin);

return fsize;
```

Program 4 *Pseudo-code* fungsi pencarian ukuran *file*

```
function htonl(
fileSizes: uint32_t in host
format)
{Mengembalikan ukuran file dari
```

```
host-byte order ke dalam
network-byte order
}

Deklarasi
array : array of byte
length: integer

Algoritma
array := BitConverter.GetBytes
(fileSizes);

if(BitConverter.IsLittleEndian)
     Array.Reverse(array);

length :=
BitConverter.ToInt32(array,0);

return ((uint32_t) length);


function ntohl(
fileSizes: uint32_t in network
format)
{Mengembalikan ukuran file dari
network-byte order ke dalam
host-byte order
}

Deklarasi
array : array of byte
length: integer

Algoritma
array := BitConverter.GetBytes
(fileSizes);

if(BitConverter.IsLittleEndian)
     Array.Reverse(array);

length :=
BitConverter.ToInt32(array,0);

return ((uint32_t) length);
```

Program 5 *Pseudo-code* fungsi konversi host-byte ke network-byte dan sebaliknya

Kita perlu mengkonversikan panjang *byte* pesan dari *host-byte order* ke dalam *network-byte order* karena setiap *environment* komputer dan sistem operasi menggunakan sistem pengkodean yang berbeda (*little-endian* atau *big-endian*). Hal ini dilakukan agar dalam pendistribusian *file* paket tersebut format ukuran *file* selalu seragam (*network-byte*) dan tidak mengalami kegagalan ketika melakukan proses ekstraksi.

Berikut adalah skema utama dalam proses pembuatan *file* paket :

```
procedure CreatePackage
(filePath1 : string,
 filePath2 : string,
  packName : string)
{Melakukan proses pembuatan
file paket dari dua file}

Deklarasi
readFile1 : BinaryReader
readFile2 : BinaryReader
writeFile1: BinaryWriter
writeFile2: BinaryWriter
data      : array of byte
data2     : array of byte
fnamelen : integer
dataSize : uint32_t

Algoritma

// Buka file pertama
readFile1  = new BinaryReader
(File.Open(filePath1,
           FileMode.Open,
           FileAccess.Read));

// Buka file kedua
readFile2  = new BinaryReader
(File.Open(filePath2,
           FileMode.Open,
           FileAccess.Read));

// Buat file data-1.pak
writeFile1 = new BinaryWriter
(File.Open("data-1.pak",
           FileMode.Create));

// Buat file data-2.pak
writeFile2 = new BinaryWriter
(File.Open("data-2.pak",
           FileMode.Create));

// Simpan ukuran kedua file
FileSizes[0] :=
  GetFileSizes(ref readFile1);
FileSizes[1] :=
  GetFileSizes(ref readFile2);

dataSize = FileSizes[0] +
FileSizes[1];

data := new byte [dataSize];
data2:= new byte
           [FileSizes[1]];
// Samarkan data agar tidak
```

```
mudah dibaca
CFB.vector =
Function.GenerateVector
(HIDDEN_DATA_KEY.Length);

data = ObfuscatingData(
readFile1.ReadBytes
((int)FileSizes[0]),filePath1);

data2 = ObfuscatingData(
readFile2.ReadBytes
((int)FileSizes[1]),filePath2);

// Tulis panjang nama arsip ke
dalam package
fnamelen = packName.Length;

// Ubah panjang pesan menjadi
network byte order
FileSizesNetFormat[0] :=
    htonl(FileSizes[0]);
FileSizesNetFormat[1] :=
    htonl(FileSizes[1]);

// Buat data-1.pak
writeFile1.Write
(collisionFirstPart,0,
 MD5_COLLISION_BLOCK_SIZE);

writeFile1.Write
((byte)fnamelen);

writeFile1.Write
(packName.ToCharArray(),0,
fnamelen);

writeFile1.Write
(FileSizesNetFormat[0]);

writeFile1.Write
(FileSizesNetFormat[1]);

// Buat data-2.pak
writeFile2.Write
(collisionSecondPart, 0,
 MD5_COLLISION_BLOCK_SIZE);

writeFile2.Write
((byte)fnamelen);

writeFile2.Write
(packName.ToCharArray(),0,
 fnamelen);

writeFile2.Write
(FileSizesNetFormat[0]);
writeFile2.Write
(FileSizesNetFormat[1]);
writeFile2.Write(data);
writeFile2.Write(data2);
writeFile2.Close();
```

Program 6 *Source-code* pembuatan paket

Pertama kita buka dulu *file* pertama dan *file* kedua dalam mode biner karena *file* masukan dapat bermacam-macam tipe. Kemudian kita buat dulu *file* keluaran, dalam hal ini telah ditentukan dengan nama `data-1.pak` dan `data-2.pak`. Setelah itu, kita cari ukuran dari masing-masing *file*, masukkan kedalam *array of* `uint32_t` (*File*`Sizes`).

Agar *file* paket tidak dapat dibaca oleh siapapun dan untuk mengurangi kecurigaan terhadap isi *file* paket, kita samarkan isi dari masing-masing *file* tersebut. Banyak cara yang dapat dilakukan untuk menyamarkan data, diantaranya dengan mengkompresi data atau mengenkripsi data dengan kunci yang terdefinisi. Pada kasus ini, penulis menyamarkan data dengan melakukan enkripsi mode CFB.

Setelah penyamaran data selesai, kita ubah dulu ukuran dari masing-masing *file* yang kita peroleh tadi menjadi *network-byte* order. Hal ini dilakukan untuk mengatasi perbedaan sistem pengkodean antara sistem operasi di setiap komputer. Kemudian kita tulis hasil proses tersebut ke `data-1.pak` dan `data-2.pak` dengan urutan seperti yang telah didefinisikan pada sub bab 3.4.3.2. Dengan demikian, *file* paket telah selesai dibuat.

Proses pembuatan *file* paket tersebut menggunakan beberapa variabel global dan data statik. Berikut adalah definisi dari masing-masing variabel dan data tersebut (dalam notasi bahasa C):

```
// Definisi data statik
#define
MD5_COLLISION_BLOCK_SIZE
(1024 / 8)

#define
MD5_COLLISION_BITMASK 0x80

#define
MD5_COLLISION_OFFSET 19

#define
```

```
HIDDEN_DATA_KEY
{0x63,0xCF,0x51}

#define
FILE_TABLE_SIZE
sizeof(uint32_t)

#define
SUBHEADER_START (1024/8)

// Definisi variabel global
uint32_t FilePos[];
uint32_t FileSizes[];
uint32_t FileSizesNetFormat[];
```

Program 7 *Pseudo-code* definisi data statik dan variabel global

Keterangan dari masing-masing variabel adalah sebagai berikut :

1. `MD5_COLLISION_BLOCK_SIZE` adalah ukuran blok *collision bytes* yang dipakai (1024/8 *bytes*)

2. `MD5_COLLISION_BITMASK` adalah *byte* kolisi yang dipakai identifikasi dalam melakukan ekstraksi *file* paket.

3. `MD5_COLLISION_OFFSET` adalah posisi tempat `MD5_COLLISION_BITMASK` berada di *file* paket

4. `HIDDEN_DATA_KEY` adalah *array of byte* yang digunakan sebagai kunci untuk menyamarkan data di dalam *file* paket.

5. *`FILE`*`_TABLE_SIZE` adalah ukuran dari panjang *file* yang disimpan. Dalam hal ini sama dengan panjang dari `uint32_t`

6. `SUBHEADER_START` adalah posisi awal penulisan data setelah blok *collision bytes*.

7. *`File`*`Pos` adalah *array of* uint32_t yang digunakan untuk menyimpan posisi awal dari *file* 1 dan *file* 2 pertama kali ditulis di dalam *file* paket.

8. *`File`*`Sizes` adalah *array of* `uint32_t` yang digunakan untuk menyimpan ukuran masing-masing *file* 1 dan *file* 2.

9. *`File`*`SizesNetFormat` adalah *array of* `uint32_t` yang digunakan untuk menyimpan ukuran kedua *file* (*file* 1 dan *file* 2) dalam *network-byte order*.

3.4.3.4 Pengekstraksian *File* Paket

Proses pengekstraksian menentukan *file* mana yang akan diekstraksi berdasarkan satu *byte* yang ada pada *magic number collision* (masing-masing *file* paket mempunyai nilai yang berbeda). Nilai *byte* yang dipakai adalah `MD5_COLLISION_OFFSET` dan `MD5_COLLISION_BITMASK`. Dua nilai ini menentukan tepat posisi dan mask dari *byte* yang saling berkolisi di dalam *file* paket. Seperti yang dapat dilihat nilai ini berada pada 128-*bytes* yang pertama.

Berikut adalah skema utama proses pengekstraksian *file* paket :

```
procedure ExtractPackage
(filePath : string)
{Melakukan proses ekstraksi
file paket}

Deklarasi
collidingByte : byte
fnamelen    : integer
fileIndex   : uint
extrSize    : uint32_t
extrPos     : uint32_t
extractBuff: array of byte
readPack    : BinaryReader
extractPack: BinaryWriter
fileName    : StringBuilder
Algoritma

// Cari dan baca byte tempat
// kolisi MD5 terjadi
readPack = new BinaryReader
(File.Open(filePath,
           FileMode.Open,
           FileAccess.Read));

readPack.BaseStream.Seek
(MD5_COLLISION_OFFSET,
 SeekOrigin.Begin);

collidingByte =
    readPack.ReadByte();

// Load nama arsip
readPack.BaseStream.Seek
(SUBHEADER_START,
 SeekOrigin.Begin);
fnamelen =
     (int) readPack.ReadByte();
fileName = new StringBuilder();
```

```
fileName.Append
(readPack.ReadChars(fnamelen));

//Load tabel arsip :: Ukuran
dua arsip yang disimpan

FileSizes[0] =
ntohl(readPack.ReadUInt32());

FileSizes[1] =
ntohl(readPack.ReadUInt32());

FilePos[0] = (uint32_t)
(SUBHEADER_START+1+
 fnamelen+FILE_TABLE_SIZE);

FilePos[1] =
     FilePos[0]+FileSizes[0];

//Baca dan ekstraksi arsip
fileIndex =
(collidingByte &
 MD5_COLLISION_BITMASK) ?
(uint) 1 : (uint)0
extrSize= FileSizes[fileIndex];
extrPos = FilePos[fileIndex];
extractBuff = new byte
                  [extrSize];
readPack.BaseStream.Seek
(extrPos +4, SeekOrigin.Begin);
readPack.Read
(extractBuff,0,(int) extrSize);
readPack.Close();

// Tulis hasil ekstraksi ke
arsip luar
if(!Directory.Exists(fileNum))
     Directory.CreateDirectory
     (fileNum);
extractPack = new BinaryWriter
(File.Open
     (fileNum+"\\"+
      fileName.ToString(),
      FileMode.Create));

extractpack.Write
(RefuscatingData(extractBuff));
extractPack.Close();
```

Program 8 *Pseudo-code* Proses Ekstraksi Arsip

Penjelasan dari algoritma proses ekstraksi adalah sebagai berikut :

1. Pertama, kita cari dulu *colliding byte* yang akan menjadi acuan dalam menentukan *file* mana yang akan diekstraksi. Kita cari pada posisi `MD5_COLLISION_OFFSET`.
2. Kemudian kita cari nama *file* hasil ekstraksi pada *file* paket ini untuk memberikan nama pada hasil ekstraksi nanti.
3. Setelah kita mencari informasi ukuran *file* yang akan diekstraksi. Karena kita belum tahu *file* mana yang akan diektraksi, maka kita simpan ukuran kedua *file*. Ukuran *file* ini kita rubah lagi menjadi *host-byte order* sehingga tidak terjadi kesalahan dalam proses selanjutnya. Kita cari juga posisi awal ektraksi dari kedua *file*.
4. Selanjutnya, adalah langkah terakhir, yaitu melakukan proses ektraksi. Kita tentukan dulu *file* mana yang akan diekstraksi, jika hasil `AND` dari *colliding byte* dengan `MD5_COLLISION_BITMASK` tidak sama dengan `MD5_COLLISION_BITMASK` maka program akan mengekstraksi *file* pertama, sebaliknya program akan mengektraksi *file* kedua.
5. Setelah kita tahu *file* mana yang akan diekstraksi, langkah selanjutnya adalah meletakkan *file* hasil ekstraksi ke dalam direktori. Proses ini hanya proses tambahan; bertujuan untuk memberikan pengetahuan bahwa hasil ektraksi dari kedua *file* adalah berbeda.

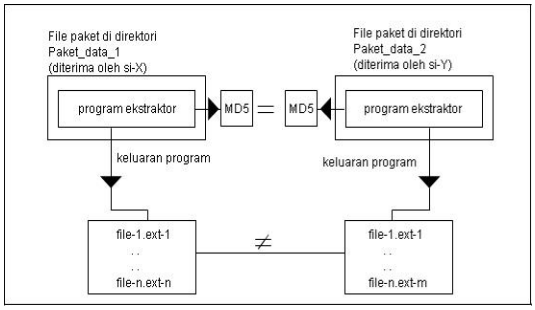
Agar hasil ekstraksi dapat langsung dijalankan, akan lebih baik ketika kita menyimpan nama *file* paket mempunyai ekstensi yang sama dengan kedua *file* yang dipaketkan.

3.4.4. Meningkatkan Serangan

Serangan ini dapat ditingkatkan dengan memaketkan *file* data tersebut dengan program ekstraksinya. Pada serangan sebelumnya terdapat restriksi bahwa kita harus meletakkan *magic number* pada awal *file* paket.

Sekali saja kita mengetahui algoritma untuk menemukan kolisi MD5 untuk setiap vektor inisialisasi, blok kolisi dapat diletakkan di setiap posisi 512-bit di dalam *file* paket. Dengan *file* paket yang membungkus program ekstraksi dan *file* data, maka si penerima tidak akan mengetahui bahwa *file* paket tersebut akan dapat mengektraksi dua *file* yang sangat berbeda. Umumnya, *file* paket yang seperti ini

(*executable*) dalam bentuk tunggal, sehingga mengurangi kecurigaan (contoh seperti *file* SFX dari WinRAR).



Gambar 8 Sekarang hanya memakai satu program ekstraktor saja untuk melakukan serangan. Kedua program ekstraktor mempunyai nilai MD5 yang sama , meskipun sebenarnya jika dijalankan akan mengekstraksi *file* yang berbeda

Namun, penulis masih belum dapat melakukan hal ini karena keterbatasan pengetahuan dan kemampuan dalam melakukan manipulasi *file* executable. Tetapi kemungkinan untuk dapat melakukan hal ini dapat terjadi. Kita hanya perlu mengutak-atik *file* executable tersebut pada bagian statik datanya. Jadi *file* paket yang akan kita ektraksi kita sisipkan tepat pada bagian statik data dan mengeset file *executable* itu untuk memulai operasi tepat pada posisi *file* paket dimasukkan.

4. Skenario serangan yang sebenarnya

Dari bab sebelumnya jelas bahwa terdapat kemungkinan untuk dapat membuat dua *file* paket ekstraktor (*executable*) yang mengandung *file-file* yang berbeda tetapi mempunyai nilai MD5 yang sama. Ambil contoh software web-browser.

Misalkan seorang pembuat paket (pemaket) di suatu perusahaan membuat *file* paket ekstraktor bermaksud untuk mendistribusikannya. Ketika pengembangan web browser selesai telah selesai, semua *file-file* web browser tersebut diberikan kepada pemaket untuk membuat kode instalasi dan menciptakan *file* paket yang siap di distribusikan. Si pemaket kemudian menulis kode tadi dan membuat *file* paket. Setelah itu, *file* paket tadi dikirimkan ke departemen pengujian untuk dites apakah *file* tersebut lolos uji dan web browser dapat di *install*. Jika semua tes berhasil dilewati, maka *file* paket tersebut dinyatakan sebagai *file* yang valid dan lemudian diberi tanda tangan digital. Departemen penguji tersebut kemudian akan menandatangani *file* paket tadi sebagai bukti bahwa *file* paket tersebut telah diuji, lulus dari semua tes, dan siap untuk didistribusikan. Kedua *file* paket tadi, nilai MD5 dan tanda tangan digitalnya, selanjutnya diletakkan pada ftp atau halaman web dari perusahaan untuk dapat didownload.

Sekarang misalkan si pemaket tadi adalah orang yang tidak jujur (iseng). Maka si pemaket tadi akan membuat pasangan *file* paket yang mempunyai nilai MD5 yang sama,satu paket berisi *file* yang asli dan satunya berisi *file* yang rusak. Ketika akan diuji, si pemaket akan memberikan *file* paket yang asli untuk diuji, sehingga dia akan mendapatkan nilai MD5 dan tanda tangan digital untuk *file* paket tersebut. Kemudian ketika *file* paket diletakkan di halaman web atau ftp dari perusahaan, maka si pemaket akan mengganti *file* paket tadi dengan *file* paket pasangannya yang berisi *file* rusak. Dengan asumsi, bahwa si pemaket mempunyai akses ke server perusahaan.

Dan jika si pemaket adalah orang yang pintar, dia akan memodifikasi sedikit software aslinya secara sembunyi-sembunyi. Software tersebut kemudian akan di *download* dan di *install*. Mengingat nilai MD5 dan tanda tangan digital tersebut tetap, dan software berjalan tidak mencurigakan, maka akan memerlukan waktu sedikit lama sampai kecacatan software tersebut terdeteksi. Semua tuduhan tentang kesalahan itu akan ditimpakan kepada departemen pengujian, karena mereka yang melakukan pengetesan dan pemberian otentikasi pada *file* paket.

Sekarang si pemaket (penyerang) dapat sebagai contoh menjual pengetahuan tentang kecacatan software kepada spammers, dan penulis kode virus, membuat halaman web pribadi yang dapat menginfeksi komputer dengan software cacat atau juga dapat meng-hack web-servers yang rentan dan memodifikasi total halaman dengan menambahkan kodenya yang dapat melakukan infeksi, dan kemudian memanfaatkan komputer yang terinfeksi untuk mengirim spam, melancarkan serangan DoS (Denial of Service), dan masih banyak lagi yang dapat dilakukan.

5. Kesimpulan dan Saran

Mungkin pada saat ini serangan X.Wang dkk. masih dianggap remeh, karena tidak dapat meng-generate *preimage collision* yang kedua. Namun, berdasarkan contoh diatas dan praktik tentang

pembuatan *file* paket ini, dapat menjadi serangan yang berbahaya. Dan akan menjadi semakin berbahaya ketika serangan X.Wang dkk dipulikasikan secara rinci. Dengan kata lain, akan sulit untuk membedakan antara serangan MD5 atau bukan pada situasi-situasi tertentu.

Dan juga, hanya dengan blok string yang saling berkolisi dapat diciptakan dua *file* paket yang mempunyai nilai MD5 yang sama namun dengan isi yang berbeda. Selanjutnya, jika algoritma ini dikembangkan, tidak menutup kemungkinan serangan ini dapat juga melakukan duplikasi terhadap *file* paket yang lain, seperti *file* .rar, .zip, .gzip, .tar dan lain sebagainya. Dengan meletakkan blok kolisi dengan benar, maka *file* paket akan dapat diduplikasi.

DAFTAR PUSTAKA

[1] Access Data On Your Radar, *White Paper : MD5 Collision The Effect on Computer Forensics, Access Data*, 2006

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

[3] http://cryptography.hyperlink.cz/2004/Hasovaci_funkce_a_cinsky_utok_MFFUK_2004.pdf

[4] http:///en.wikipedia.org/wiki/MD5

[5] *http://msdn.microsoft.com*

[6] http://www.ietf.org/rfc/rfc1321.txt

[7] http://www.stachliu.com/md5coll.c

[8] http://www.ipa.go.jp/security/rfc/RFC1321EN.html

[9] Marc Stevens, Arjen Lenstra, Benne de Wrger, Target Collisions for MD5 and *Colliding* X.509 Certificates for Different Identities, 2006.

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

[11] Mikle, Ondrej. *Practical Attack on Digital Signatures Using MD5 Collision*. Department of Software Engineering at Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic. 2004

[12] Philip Hawkes, Michael Paddon, and Gregory G.Rose, *Musings on The Wang et al.MD5 Collision*,2006.

[13] Praveen Gauravaram, William Millan, and Juanma Gonzales Neito, *Some Thought on Collision Attacks in The Hash Functions MD5, SHA-0, SHA-1*, 2006.

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

[15] 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

[16] Yu Sasaki, Yusuke Naito, Jun Yajima, Takeshi Shimoyama, Noboru Kunihiro Kazuo Ohta, *How to Construct Sufficient Condition in Searching Collisions of MD5*, 2006

Note : Program dan source code dapat didownload di *http://students.if.itb.ac.id/~if14063/md5_coll/*