Perbandingan Performansi dan Tingkat Keamanan dari Implementasi MD5, MD6 dan WHIRLPOOL

Roy Indra Haryanto 13508026
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
royindra@students.itb.ac.id

***Abstract*—Fungsi *hash* adalah fungsi yang menerima masukkan *string* yang panjangnya sembarang dan mengkonversinya menjadi *string* keluaran yang panjangnya tetap atau *fixed*. Fungsi *hash* dapat diaplikasikan dalam kehidupan sehari-hari. Salah satu contoh aplikasi fungsi *hash* adalah verifikasi sebuah salinan dokumen dengan dokumen aslinya. Verifikasi salinan dokumen tersebut bertujuan untuk mengetahui apakah salinan dokumen masih sama atau sudah mengalami perubahan (secara sengaja atau tidak sengaja) jika dibandingkan dengan dokumen aslinya.**

Pada makalah ini akan dibahas tentang tiga algoritma fungsi hash yang cukup terkenal, yaitu MD5, MD6, WHIRLPOOL. Akan dibahas bagaimana cara kerja algoritma tersebut, implemetasi dari ketiga algoritma tersebut dan bagaimana perbadingan hasil implementasinya.

***Index Terms*—Fungsi Hash, MD5, MD6, WHIRLPOOL**

I. PENDAHULUAN

Sebuah fungsi hash dalam kriptografi adalah fungsi deterministik yang mengambil sejumlah blok data dan mengembalikan string dengan bpanjang bit tetap, yang disebut dengan nilai hash, sehingga perubahan baik yang disengaja maupun tidak disengaja kepada data akan mengubah nilai hash. Data yang perlu dikodekan disebut pesan, dan nilai hash kadang-kadang disebut *message digest*.

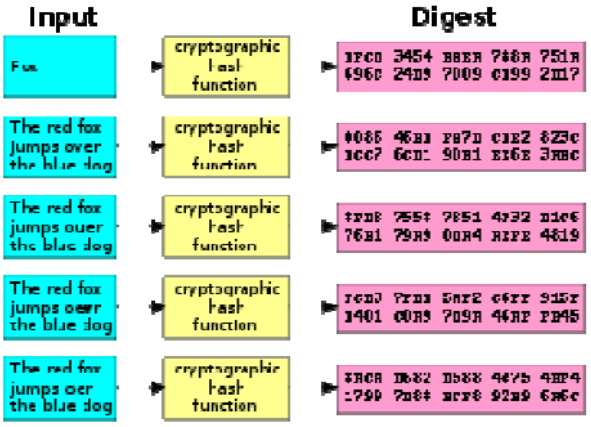
Persamaan fungsi hash dapat dituliskan dengan persamaan berikut :

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

dengan,

M = pesan berukuran sembarang.

h = nilai hash (*hash value*) atau pesan ringkas (*message-digest*) dengan ukuran tetap.



Gambar 1 – Skema Fungsi Hash

Dalam kriptografi, keberadaan fungsi hash yang ama secara kriptografik sangatlah penting. Fungsi hash semacam ini memiliki properti – properti sebagai berikut.

- *Collision resistance*: seseorang seharusnya tidak dapat menemukan dua pesan berbeda, sebut saja M dan M' sedemikian sehingga H(M) = H(M') (terjadi kolisi)
- *First preimage resistance* : seseorang yang diberikan hasil hash h seharusnya tidak dapat menemukan M dimana H(M) = h. Salah satu contoh mengapa hal ini penting yakni pada umumnya sandi lewat pengguna disimpan dalam bentuk hash. Jika seseorang yang memiliki akses pada data sandi lewat yang telah di*hash* maka seharusnya ia tidak bisa memperoleh sandi lewat yang asli dari data tersebut. Akan tetapi, hal ini mungkin terjadi jika properti tidak dipenuhi.
- *Second preimage resistance* : seseorang yang memiliki pesan M seharusnya tidak dapat memperoleh pesan M' dimana M tidak sama dengan M' tetapi H(M) = H(M'). Properti ini merupakan implikasi dari *collision resistance*.

Selain properti yang telah disebutkan di atas, ada dua properti lain lagi yang ditujukan untuk fungsi hash yang termasuk ke dalam *keyed hash function*, yakni :

- *Pseudorandomness* : seseorang seharusnya tidak bisa membedakan keluaran fungsi hash dari fungsi random murni. Properti ini secara langsung mengimplikasikan unpredictability, tetapi tidak sebaliknya
- *Unpredictability* : seseorang yang diberikan akses terhadap suatu fungsi hash seharusnya tidak dapat menebak keluarannya tanpa sebelumnya mengetahui pesan yang akan dihash. Dengan kata lain, seseorang dengan hak akses tersebut tidak mungkin mendapatkan pasangan nilai (M, h) di mana H(M) = h tanpa mengetahui nilai M.

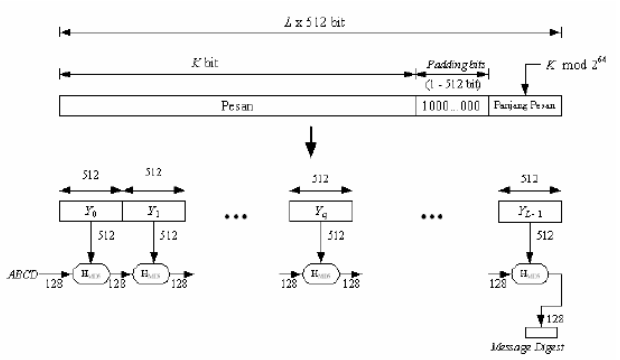
II. LANDASAN TEORI

A. Algoritma MD5

MD5 didasarkan pada mode operasi iteratif berantai yang umumnya disebut dengan konstruksi Merkle-Damgard. Konstruksi Merkle-Damgard pada dasarnya menggunakan fungsi kompresi $f:\{0,1\}^{n+\ell} \rightarrow \{0,1\}n$, atau sebuah block cipher E yang dibuat untuk berperilaku sebagai fungsi kompresi melalui transformasi Davies-Meyer: $f(x,y)=Ex(y) \oplus y$. Jika f merupakan sebuah fungsi kompresi, maka konstruksi Merkle-Damgard yang menggunakan fungsi kompresi ini, misalkan saja namanya MD_f, dimulai dengan menambahkan pesan masukan m agar memiliki panjang yang merupakan kelipatan ℓ, kemudian mengambil vektor inisialisasi IV berukuran n-bit. Ia kemudian berjalan secara sekuensiaol melalui ℓ-bit message chunks, mulai dari chunk pertama, dan berakhir setelah pemrosesan chunk terakhir. Berikut pseudocode algoritma tersebut.

Input : $m = m_1 \| m_2 \| \ldots \| m_t$, di mana $\|m_i\| = \ell$ untuk semua i. Output: *message digest*, h.
$y_0 \leftarrow IV$ for i ← 1 to t $y_i \leftarrow f(y_{i-1},m_i)$ $h \leftarrow y_t$

Ilustrasi operasi di atas dapat dilihat pada Gambar 2. Gambar 2 tersebut merupakan contoh konstruksi Merkle-Damgard pada MD5.



Gambar 2 – Konstruksi Merkle-Damgard pada MD5

Algoritma MD5 dapat dibagi menjadi empat tahap, yakni penambahan bit-bit pengganjal (*padding bits*), penambahan nilai panjang pesan semula, inisialisasi penyangga (*buffer*) *MD*, dan pengolahan pesan dalam blok berukuran 512 bit. Berikut penjelasan masing-masing tahap tersebut.

Penambahan Padding bits

Pesan ditambah dengan sejumlah bit pengganjal sedemikian sehingga panjang pesan dalam satuan bit kongruen dengan 448 modulo 512. Jika panjang pesan ternyata 448 bit, maka pesan tersebut ditambah 512 bit sehingga menjadi 960 bit. Jadi, panjang bit-bit pengganjal adalah antara 1 sampai 512. Bit-bit pengganjal terdiri dari sebuah bit 1 diikuti dengan bit 0 untuk sisanya.

Penambahan Nilai Panjang Pesan

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 telah menjadi kelipatan 512 bit.

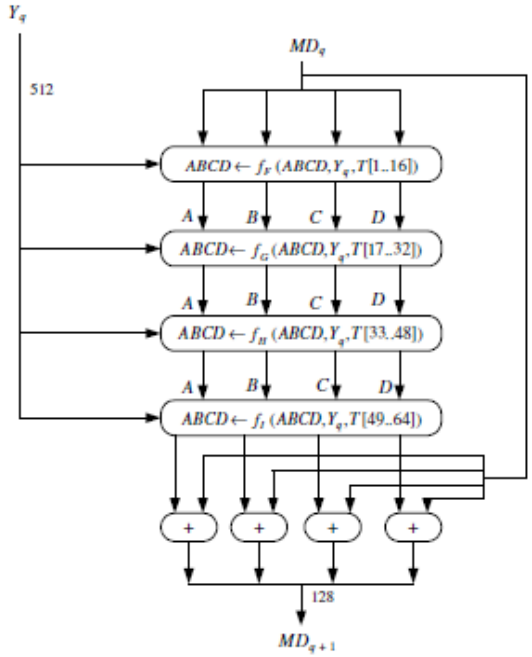
Inisialisasi Penyangga MD

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

A = 01234567
B = 89ABCDEF
C = FEDCBA98
D = 76543210

Pengolahan Pesan dalam Blok Berukuran 512 bit

Pesan dibagi menjadi *L* buah blok yang masing-masing panjangnya 512 bit (*Y*0 sampai *YL* – 1). Setiap blok 512-bit diproses bersama dengan penyangga *MD* menjadi keluaran 128-bit, dan ini disebut proses *H*MD5. Gambaran proses H_{MD5} diperlihatkan pada Gambar 3.



Gambar 3 – Proses H_{MD5}

Pada Gambar 3, Yq 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_{MD5} ke-q. Pada awal proses, MDq berisi nilai inisialisasi penyangga MD. Proses H_{MD5} terdiri dari 4 buah putaran. Fungsi-fungsi f_F, f_G, f_H, dan f_I masing-masing berisi 16 kali operasi dasar terhadap masukan, setiap operasi dasar menggunakan nilai Ti sesuai Tabel 1. Perbedaan masing-masing fungsi f_F, f_G, f_H, dan f_I sendiri dapat dilihat pada Tabel 2.

Tabel 1 – Nilai Ti

T[1] = D76AA478	T[33] = FFFA3942
T[2] = E8C7B756	T[34] = 8771F681
T[3] = 242070DB	T[35] = 69D96122
T[4] = C1BDCEEE	T[36] = FDE5380C
T[5] = F57C0FAF	T[37] = A4BEEA44
T[6] = 4787C62A	T[38] = 4BDECFA9
T[7] = A8304613	T[39] = F6BB4B60
T[8] = FD469501	T[40] = BEBFBC70
T[9] = 698098D8	T[41] = 289B7EC6
T[10] = 8B44F7AF	T[42] = EAA127FA
T[11] = FFFF5BB1	T[43] = D4EF3085
T[12] = 895CD7BE	T[44] = 04881D05
T[13] = 6B901122	T[45] = D9D4D039
T[14] = FD987193	T[46] = E6DB99E5
T[15] = A679438E	T[47] = 1FA27CF8
T[16] = 49B40821	T[48] = C4AC5665
T[17] = F61E2562	T[49] = F4292244
T[18] = C040B340	T[50] = 432AFF97
T[19] = 265E5A51	T[51] = AB9423A7
T[20] = E9B6C7AA	T[52] = FC93A039
T[21] = D62F105D	T[53] = 655B59C3
T[22] = 02441453	T[54] = 8F0CCC92
T[23] = D8A1E681	T[55] = FFEFF47D
T[24] = E7D3FBCB	T[56] = 85845DD1
T[25] = 21E1CDE6	T[57] = 6FA87E4F
T[26] = C33707D6	T[58] = FE2CE6E0
T[27] = F4D50D87	T[59] = A3014314
T[28] = 455A14ED	T[60] = 4E0811A1
T[29] = A9E3E905	T[61] = F7537E82
T[30] = FCEFA3F8	T[62] = BD3AF235
T[31] = 676F02D9	T[63] = 2AD7D2BB
T[32] = 8D2A4C8A	T[64] = EB86D391

Tabel 2 – 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	$I(b, c, d)$	$c \oplus (b \wedge \sim d)$

Operasi dasar *MD5* dapat ditulis dengan sebuah persamaan sebagai berikut:

$$a \leftarrow b + \text{CLS}_s(a + g(b, c, d) + X[k] + T[i]) \qquad (2)$$

dengan,

a, b, c, d = empat buah peubah penyangga 32 bit

g = salah satu fungsi F, G, H, I

CLS_s = *circular left shift* sebanyak s bit

X[k] = kelompok 32-bit ke-k dari blok 512 bit message ke-q, k bernilai antara 0 sampai 15

T[i] = elemen Tabel T ke-i (32 bit)

Untuk nilai shift yang digunakan pada setiap putaran akan mengikuti aturan berikut :

- Untuk i antara 0 sampai 15, nilai shift berulang mengikuti pola 7, 12, 17, 22
- Untuk i antara 16 sampai 31, nilai shift berulang mengikuti pola 5, 9, 14, 20
- Untuk i antara 32 sampai 47, nilai shift berulang mengikuti pola 4, 11, 16, 23
- Untuk i antara 48 sampai 63, nilai shift berulang mengikuti pola 6, 10, 15, 21

B. Algoritma MD6

Algoritma MD6 menerima masukan berupa sebuah pesan M dengan panjang positif m dan panjang keluaran d yang berkisar antara 1 sampai dengan 512 bit. Ada pula masukan opsional yakni key K, parameter mode L, dan jumlah ronde operasi r. Secara garis besar, algoritmanya mengikuti langkah berikut :

1. Inisialisasi nilai $\ell = 0$, $M_0 = M$, dan $m_0 = m$. ℓ adalah *current level*.
2. Kemudian, untuk setiap level, lakukan proses berikut.
   - Jika $\ell = L + 1$, panggil fungsi SEQ($M_{\ell-1}$,d,K,L,r) sebagai keluaran fungsi hash.
   - Panggil fungsi PAR($M_{\ell-1}$,d,K,L,r,ℓ) dan masukkan hasilnya ke dalam M_ℓ.
   - Kemudian, jika panjang M_ sama dengan c word, kembalikan d bit terakhir dari M_ℓ sebagai keluaran fungsi hash. Secara default, nilai c adalah 16 word.

Perlu diperhatikan bahwa jika fungsi SEQ dipanggil, maka fungsi PAR tidak akan dipanggil.

Fungsi SEQ

Fungsi SEQ($M_{\ell-1}$,d,K,L,r) merupakan fungsi hash sekuensial yang mirip dengan konstruksi Merkle-Damgard. Fungsi ini tidak dipanggil pada pengaturan *default* yakni nilai L = 64. Akan tetapi, misalnya jika nilai L = 0, fungsi ini dipanggil.

Fungsi PAR

Fungsi PAR($M_{\ell-1}$,d,K,L,r,ℓ) merupakan fungsi yang melakukan operasi kompresi paralel yang membentuk level ℓ pohon dari level ℓ-1.

Fungsi Kompresi *f*

Fungsi kompresi f memetakan blok 64 word menjadi 16 word pada fungsi SEQ maupun PAR. Fungsi ini menerima masukan jumlah ronde r dan array N[0..n-1] dengan n = 89 word yang terdiri dari 64 word data dan 25 word informasi tambahan. Secara default, nilai r dihitung dengan persamaan :

$$r = 40 + d/4 \quad (3)$$

dengan d adalah panjang bit keluaran has yang rentang nilainya 0 – 512.

Fungsi kompresi f ini memiliki langkah sebagai berikut :

1. Pertama, hitung t = rc (setiap ronde memiliki c = 16 putaran).
2. Array A[0..t+n-1] merupakan array dari (t+n) wrod
3. Inisialisasi nilai A[0..n-1] dengan masukan N[0..n-1]
4. Kemudian, dilakukan loop untuk perhitungan elemen A ke-i. i memiliki rentang nilai n-(t+n-1). Proses perhitungannya yakni dengan persamaan (4).

$$\begin{aligned} x &= S_{-n} \oplus A_{i-n} \oplus A_{i-t0} \\ x &= x \oplus (A_{i-t1} \wedge A_{i-t2}) \oplus (A_{i-t3} \wedge A_{i-t4}) \\ x &= x \oplus (x >> r_{i-n}) \\ A_i &= x \oplus (x << \ell_{i-n}) \end{aligned} \quad (4)$$

dengan konstanta sebagai berikut :

$\mathbf{S_{i-n}}$ = konstanta ronde

$\mathbf{t_0,t_1,t_2,t_3,t_4}$ = posisi *tap*

$\mathbf{r_{i-n}}$ = banyaknya shift kanan

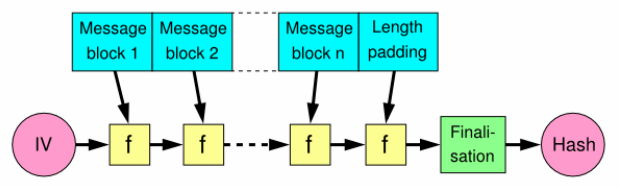
ℓ_{i-n} = banyaknya shift kiri

5. Nilai A[t+n-c..t+n-1] merupakan keluaran dari fungsi kompresi yang memiliki panjang c = 16 word

C. Algoritma WHIRLPOOL

Whirlpool adalah fungsi *hash* satu-arah yang dirancang oleh Vincent Rijmen dan Paulo S. L. M. Barreto. Whirlpool beroperasi pada *message* sembarang dengan panjang kurang dari 2_{256} bit, dan menghasilkan keluaran *message digest* dengan panjang tetap, yaitu 512 bit. *Message digest* yang dihasilkan adalah angka-angka dalam format heksadesimal. Karena satu angka heksadesimal mempunyai panjang 4 bit, panjang *message digest* yang dihasilkan 512 / 4 = 128 angka.

Whirlpool adalah fungsi *hash* yang berbasis pada *block cipher*. Whirlpool menggunakan konstruksi Merkle-Damgard sehingga fungsi ini dapat mengubah pesan masukan dengan panjang sembarang dan menghasilkan *message digest* dengan panjang tetap.



Gambar 4 – Konstruksi Merkle-Damgard

Dapat dilihat dari gambar di atas bahwa *message* masukan dengan panjang sembarang dipecah-pecah menjadi blok-blok dengan panjang sama. Panjang blok untuk Whirlpool adalah 512. Blok-blok *message* tersebut selanjutnya akan diproses secara sekuensial dengan fungsi kompresi f.

Dapat kita katakan bahwa secara garis besar, langkah-langkah pembuatan message digest adalah:

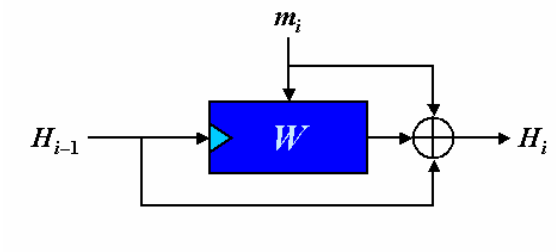
1. Inisialisasi IV (initialization vector). Dalam Whirlpool, IV berukuran 512 bit dan semua bit-nya berisi bit "0".
2. Prosedur finalization dengan cara menambahkan padding bits dan penambahan nilai panjang message.
3. Pengolahan pesan dengan fungsi f.

Penambahan Padding Bits

Agar blok *message* yang menjadi masukan fungsi f selalu mempunyai panjang tetap (512 bit), dilakukan prosedur *padding*. *Message* akan ditambah dengan sejumlah bit-bit pengganjal sedemikian sehingga panjang pesan (dalam satuan bit) kongruen dengan 256 modulo 512. Bit-bit pengganjal yang ditambahkan tersebut terdiri dari sebuah bit "1" yang diikuti dengan bit-bit "0". 256 bit sisa dari blok *message* akan diisi dengan panjang *message* masukan. Dengan demikian, setelah dilakukan prosedur *finalization* ini, akan didapat *message* dengan panjang kelipatan 512 bit.

Fungsi Kompresi

Fungsi f yang digunakan dalam Whirlpool adalah fungsi kompresi Miyaguchi-Preneel. Skema fungsi Miyaguchi-Preneel dapat dilihat di bawah ini:



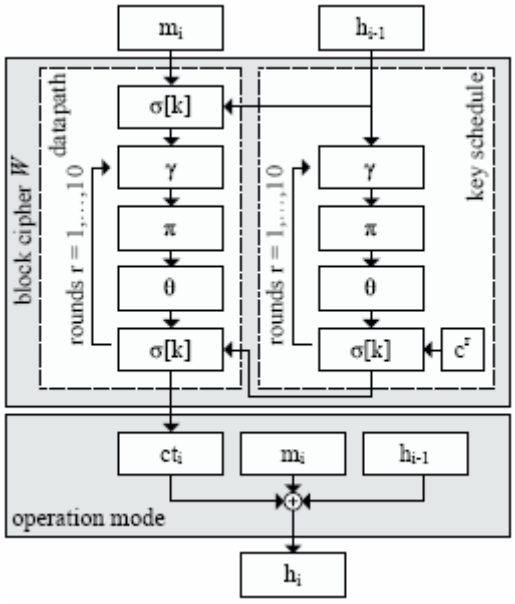
Gambar 5 – Fungsi Miyaguchi-Preneel

Fungsi ini menerima dua masukan: blok *message* yang *current* (m_i) dan hasil fungsi sebelumnya (H_{i-1}). Dua masukan in kemudian diproses berbasis *block cipher* 512 bit W. *Block cipher* W dapat dikatakan mengenkripsi m_i dengan kunci H_{i-1}. Hasil dari *block cipher* W kemudian di-XOR-kan kembali dengan dua masukan tadi. Hasil peng-XOR-an tersebutlah yang akan menjadi masukan fungsi selanjutnya.

Block Cipher Internal W

Block cipher yang digunakan dalam Whirlpool sangat mirip dengan algoritma AES. W adalah *block cipher* 512 bit dan menggunakan kunci dengan panjang 512 bit pula. Yang berperan sebagai plainteks dalam hal ini adalah blok *message* yang ke-i (m_i), sedangkan yang berperan sebagai kunci (*cipher key*) adalah hasil dari fungsi *hash* sebelumnya (H_{i-1}).

Block cipher W secara umum dapat dibagi menjadi dua bagian: *datapath* dan *key schedule*. Gambar di bawah ini akan memperjelas aliran data dalam *block cipher* W.



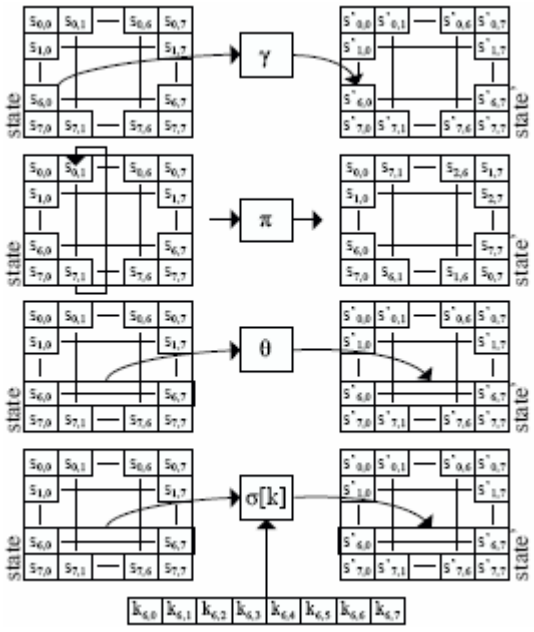
Gambar 6 – Aliran Data Whirlpool

Datapath menerima masukan blok *message* m_i dengan panjang 512 bit dan memprosesnya secara iteratif dengan melakukan *round transformations* sebanyak 10 kali. Dapat dilihat pada gambar 6 bahwa setiap ronde pada *datapath* memerlukan *round key* yang dapat diturunkan dari proses *key schedule*. Setelah 10 ronde telah dilakukan, akan dihasilkan keluaran cipherteks ct_i.

Proses *key schedule* pada dasarnya menggunakan 10 ronde *round transformations* yang sama dengan *datapath*. Perbedaannya adalah *key schedule* menggunakan *round constants* sebagai masukannya.

Round Transformation

Gambaran mengenai *round transformations* dapat dilihat pada gambar di bawah ini:



Gambar 7 – Keempat Round Transformation

Penjelasan dari keempat *round transformations* tersbut sebagai berikut:

1. γ adalah langkah non linear dimana setiap *byte* dari masukan akan disubstitusikan ke dalam S-Box.
2. π adalah permutasi putaran (*cyclical permutation*) dimana *byte-byte* dari kolom j dirotasikan ke bawah sebanyak j kali.
3. θ adalah langkah difusi linear dimana masukan dilipatgandakan dengan matriks konstatnta.
4. σ[k] penambahan kunci (*key addition*) dimana pada *datapath*, kunci didapatkan dari hasil *key schedule*, sedangkan pada *key schedule* itu sendiri, kunci didapatkan dari *round constants*.

Dapat dilihat pula dari gambar 7 bahwa satu ronde transformasi ρ[k] dari *W* dapat dinyatakan secara matematis sebagai:

$$\rho[k] \equiv \sigma[k] \circ \theta \circ \pi \circ \gamma \qquad (5)$$

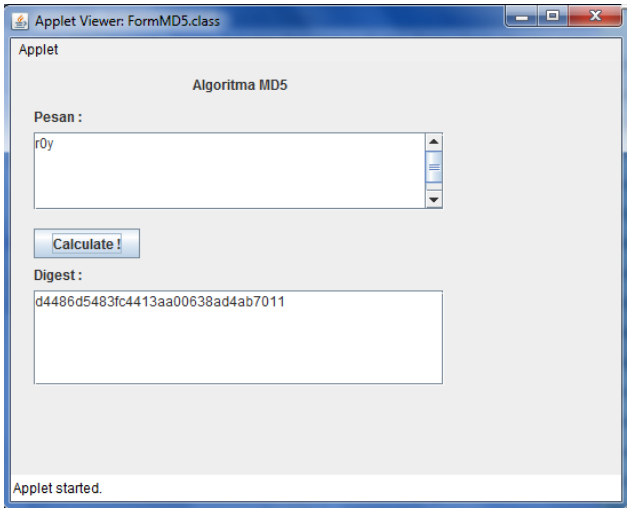
III. Implementasi dan Analisis

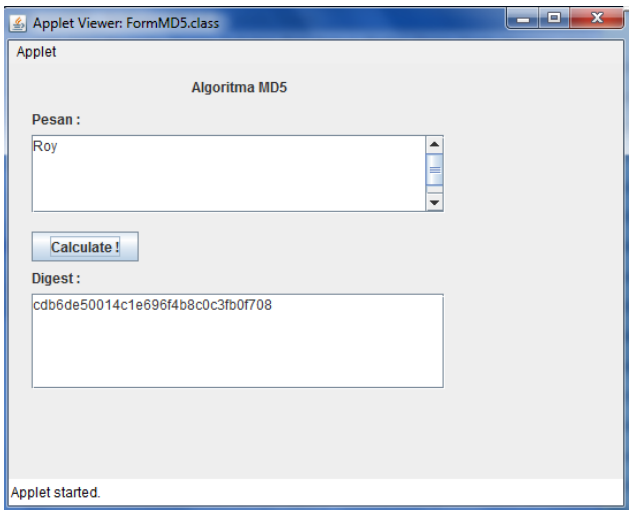
A. Implementasi MD5

Implementasi dilakukan dalam Java Applet. Berikut ini adalah antarmuka dan hasil uji coba dari implementasi algoritma MD5.

Uji Coba Kesensitifan Fungsi Hash

Dilakukan uji coba dengan data test berupa string "Roy" dan "r0y". Hasilnya adalah sebagai berikut :



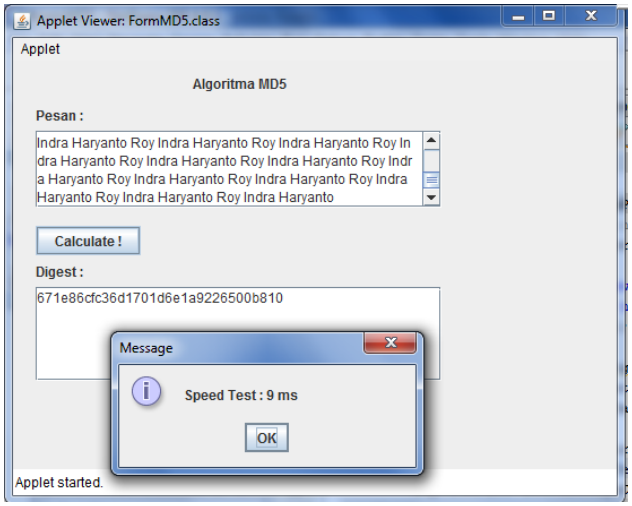


Gambar 8 – Hasil Ujicoba Kesensitifan Fungsi Hash

Dari gambar di atas, dapat dilihat ternyata fungsi hash MD5 memang sensitif dengan perubahan walaupun sedikit pada pesan.

Uji Coba Kecepatan Kalkulasi

Dengan data test berupa string "Roy Indra Haryanto" yang ditulis berulang kali yang menghasilkan jumlah 6.726 karakter, didapat hasil sebagai berikut :



Gambar 9 – Hasil Ujicoba Kecepatan Kalkulasi MD5

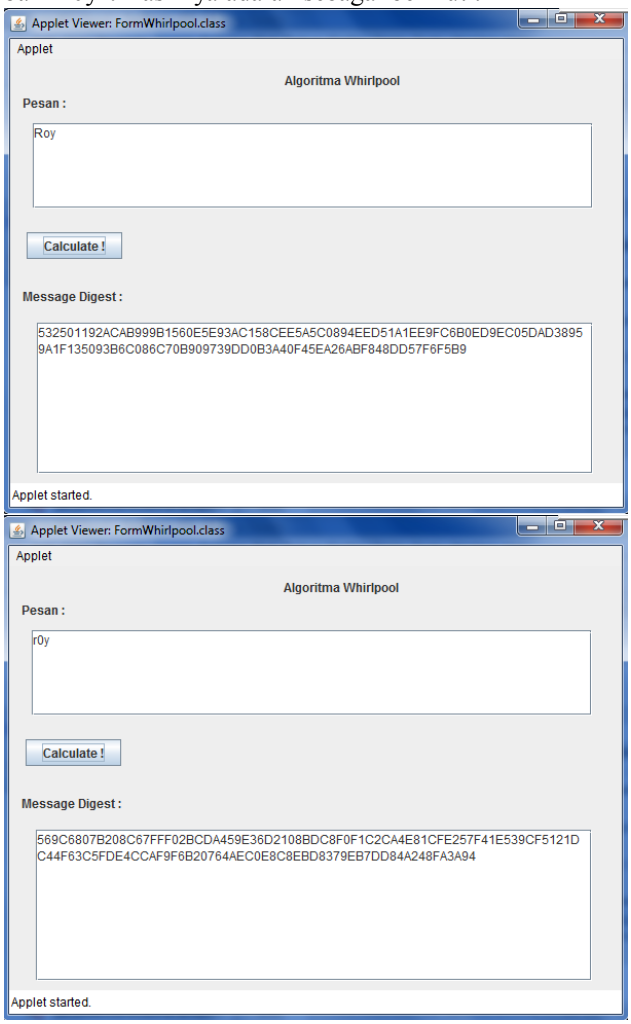
Dari gambar di atas, dapat dilihat bahwa message digest dihasilkan dalam waktu 9 millisecond.

B. Implementasi WHIRLPOOL

Implementasi dilakukan dalam Java Applet. Berikut ini adalah antarmuka dan hasil uji coba dari implementasi algoritma WHIRLPOOL.

Uji Coba Kesensitifan Fungsi Hash

Dilakukan uji coba dengan data test berupa string "Roy" dan "r0y". Hasilnya adalah sebagai berikut :

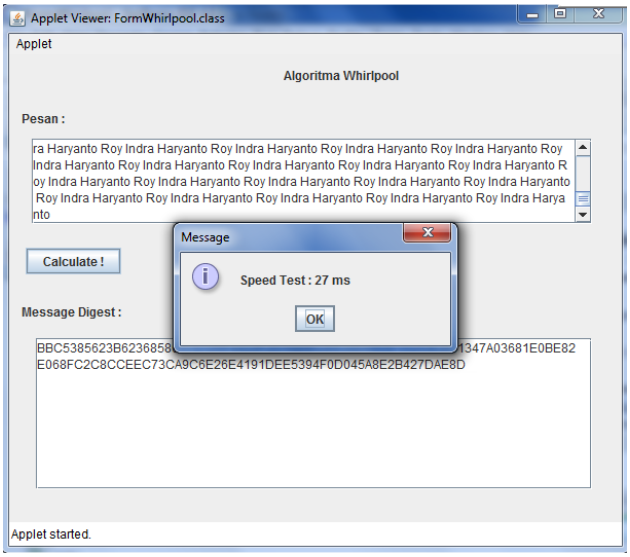


Gambar 10 – Hasil Ujicoba Kesensitifan WHIRLPOOL

Dari gambar di atas, dapat dilihat ternyata fungsi hash WHIRLPOOL memang sensitif dengan perubahan walaupun sedikit pada pesan.

Uji Coba Kecepatan Kalkulasi

Dengan data test yang sama seperti pada ujicoba MD5, berupa string "Roy Indra Haryanto" yang ditulis berulang kali yang menghasilkan jumlah 6.726 karakter, didapat hasil sebagai berikut :



Gambar 11 – Hasil Ujicoba Kecepatan Kalkulasi WHIRLPOOL

Dari gambar di atas, dapat dilihat bahwa message digest dihasilkan dalam waktu 27 millisecond.

C. Implementasi MD6

Untuk algoritma MD6, penulis cukup mengalami kesulitan dalam melakukan implementasinya. Oleh karena itu, penulis melakukan eksperimen dengan melakukan uji coba terhadap source code yang disediakan di situs

http://groups.csail.mit.edu/cis/md6/downloads.html

yang merupakan source code asli dalam bahasa C yang disubmit ke *NIST SHA-3 competition*.

Berikut ini adalah kutipan hasil ujicoba yang dicantumkan dalam dokumentasi report terhadap MD6. Ujicoba dilakukan dengan data test berupa string "abc"

```
> md6sum -r5 -I -Mabc
-r5
-- Mon Aug 04 18:28:03 2008
-- d = 256 (digest length in bits)
-- L = 64 (number of parallel passes)
-- r = 5 (number of rounds)
  -- K = '' (key)
  -- k = 0 (key length in bytes)
  MD6 compression function computation (level 1,
index 0):
  Input (89 words):
  A[ 0] = 7311c2812425cfa0 Q[0]
  A[ 1] = 6432286434aac8e7 Q[1]
  A[ 2] = b60450e9ef68b7c1 Q[2]
  A[ 3] = e8fb23908d9f06f1 Q[3]
  A[ 4] = dd2e76cba691e5bf Q[4]
  A[ 5] = 0cd0d63b2c30bc41 Q[5]
  A[ 6] = 1f8ccf6823058f8a Q[6]
  A[ 7] = 54e5ed5b88e3775d Q[7]
  A[ 8] = 4ad12aae0a6d6031 Q[8]
  A[ 9] = 3e7f16bb88222e0d Q[9]
  A[ 10] = 8af8671d3fb50c2c Q[10]
  A[ 11] = 995ad1178bd25c31 Q[11]
  A[ 12] = c878c1dd04c4b633 Q[12]
  A[ 13] = 3b72066c7a1552ac Q[13]
  A[ 14] = 0d6f3522631effcb Q[14]
  A[ 15] = 0000000000000000 key K[0]
  A[ 16] = 0000000000000000 key K[1]
  A[ 17] = 0000000000000000 key K[2]
```

```
  A[ 18] = 0000000000000000 key K[3]
  A[ 19] = 0000000000000000 key K[4]
  A[ 20] = 0000000000000000 key K[5]
  A[ 21] = 0000000000000000 key K[6]
  A[ 22] = 0000000000000000 key K[7]
  A[ 23] = 0100000000000000 nodeID U = (ell,i) =
(1,0)
  A[ 24] = 00054010fe800100 control word V =
(r,L,z,p,keylen,d) = (5,64,1,4072,0,256)
  A[ 25] = 6162630000000000 data B[ 0] input
message word 0
  A[ 26] = 0000000000000000 data B[ 1] padding
  A[ 27] = 0000000000000000 data B[ 2] padding
  A[ 28] = 0000000000000000 data B[ 3] padding
  A[ 29] = 0000000000000000 data B[ 4] padding
  A[ 30] = 0000000000000000 data B[ 5] padding
  A[ 31] = 0000000000000000 data B[ 6] padding
  A[ 32] = 0000000000000000 data B[ 7] padding
  A[ 33] = 0000000000000000 data B[ 8] padding
  A[ 34] = 0000000000000000 data B[ 9] padding
  A[ 35] = 0000000000000000 data B[10] padding
  A[ 36] = 0000000000000000 data B[11] padding
  A[ 37] = 0000000000000000 data B[12] padding
  A[ 38] = 0000000000000000 data B[13] padding
  A[ 39] = 0000000000000000 data B[14] padding
  A[ 40] = 0000000000000000 data B[15] padding
  A[ 41] = 0000000000000000 data B[16] padding
  A[ 42] = 0000000000000000 data B[17] padding
  A[ 43] = 0000000000000000 data B[18] padding
  A[ 44] = 0000000000000000 data B[19] padding
  A[ 45] = 0000000000000000 data B[20] padding
  A[ 46] = 0000000000000000 data B[21] padding
  A[ 47] = 0000000000000000 data B[22] padding
  A[ 48] = 0000000000000000 data B[23] padding
  A[ 49] = 0000000000000000 data B[24] padding
  A[ 50] = 0000000000000000 data B[25] padding
  A[ 51] = 0000000000000000 data B[26] padding
  A[ 52] = 0000000000000000 data B[27] padding
  A[ 53] = 0000000000000000 data B[28] padding
  A[ 54] = 0000000000000000 data B[29] padding
  A[ 55] = 0000000000000000 data B[30] padding
  A[ 56] = 0000000000000000 data B[31] padding
  A[ 57] = 0000000000000000 data B[32] padding
  A[ 58] = 0000000000000000 data B[33] padding
  A[ 59] = 0000000000000000 data B[34] padding
  A[ 60] = 0000000000000000 data B[35] padding
  A[ 61] = 0000000000000000 data B[36] padding
  A[ 62] = 0000000000000000 data B[37] padding
  A[ 63] = 0000000000000000 data B[38] padding
  A[ 64] = 0000000000000000 data B[39] padding
  A[ 65] = 0000000000000000 data B[40] padding
  A[ 66] = 0000000000000000 data B[41] padding
  A[ 67] = 0000000000000000 data B[42] padding
  A[ 68] = 0000000000000000 data B[43] padding
  A[ 69] = 0000000000000000 data B[44] padding
  A[ 70] = 0000000000000000 data B[45] padding
  A[ 71] = 0000000000000000 data B[46] padding
  A[ 72] = 0000000000000000 data B[47] padding
  A[ 73] = 0000000000000000 data B[48] padding
  A[ 74] = 0000000000000000 data B[49] padding
  A[ 75] = 0000000000000000 data B[50] padding
  A[ 76] = 0000000000000000 data B[51] padding
  A[ 77] = 0000000000000000 data B[52] padding
  A[ 78] = 0000000000000000 data B[53] padding
  A[ 79] = 0000000000000000 data B[54] padding
  A[ 80] = 0000000000000000 data B[55] padding
  A[ 81] = 0000000000000000 data B[56] padding
  A[ 82] = 0000000000000000 data B[57] padding
  A[ 83] = 0000000000000000 data B[58] padding
  A[ 84] = 0000000000000000 data B[59] padding
  A[ 85] = 0000000000000000 data B[60] padding
  A[ 86] = 0000000000000000 data B[61] padding
  A[ 87] = 0000000000000000 data B[62] padding
  A[ 88] = 0000000000000000 data B[63] padding
  Intermediate values:
  A[ 89] = 027431e67f2b19cf
  A[ 90] = 0d990f6680e90d20
  A[ 91] = f27bc123aa282635
  A[ 92] = f90ca91b7fd9c62c
  A[ 93] = 85139f55bd354f15
  A[ 94] = eb6b874532011a19
  A[ 95] = 7b04461ba005d2fc
  A[ 96] = c7db19c96ca9abc7
  A[ 97] = b723400f04c813c4
  A[ 98] = c22c98f63ef66335
  A[ 99] = 42a2cbb64372fc40
  A[ 100] = e52aeb1d587b9012
  A[ 101] = 9ea7a2d571275633
  A[ 102] = 7e99d0316f65addd
  A[ 103] = 72f2b2f2fd1fe6ec
  A[ 104] = 478df0ec797df153
  A[ 105] = 3b9efe3b34add3eb
  A[ 106] = f0155b54e33fa5cc
  A[ 107] = b3b80e2309548fa4
  A[ 108] = b5ef06df65e727d7
  A[ 109] = ef08a1b814d205a0
  A[ 110] = 367b2caf36cc81c6
  A[ 111] = 343a0cf5b903d13e
  A[ 112] = b4f9c1e7889e619e
  A[ 113] = da463bc1b64240ad
  A[ 114] = 10401204b0e3df85
  A[ 115] = 4877a679f7db2705
  A[ 116] = e2ff7c19283b650d
  A[ 117] = 7e20b510048c8b81
  A[ 118] = 2ec6248f95796fcd
  A[ 119] = 0c87c7f9e1056f74
  A[ 120] = 5e20250caa5b4a43
  A[ 121] = 6e44865c042e3829
  A[ 122] = 9529fbc6155a6a6d
  A[ 123] = c44d6a63399d5e4f
  A[ 124] = 04ead78d74346144
  A[ 125] = 259b97c077a30362
  A[ 126] = d185200a80400541
  A[ 127] = b9a8bba23413f53c
  A[ 128] = a439ca3d5839a512
  A[ 129] = d2be51693c027782
  A[ 130] = 94c0710d616da4c0
  A[ 131] = 55e60934532be3b6
  A[ 132] = a6e5b044f10f495d
  A[ 133] = c2a4ba0dd30863e0
  A[ 134] = abfa7c9a10170f52
  A[ 135] = c55ba748fdfdcaaa
  A[ 136] = 9e0f8e2fbf4645e7
  A[ 137] = 21b0d68b36a65ab3
  A[ 138] = 24e5578b36da9478
  A[ 139] = 58446db406441646
  A[ 140] = 1be8e6525fc16819
  A[ 141] = e84464fb02c603b9
  A[ 142] = a14656016a6def39
  A[ 143] = 9b2b76febbe7de1f
  A[ 144] = 79eda3eb98f56b99
  A[ 145] = 0d4ce347389fbe8d
  A[ 146] = 0e51deba9751e9ac
  A[ 147] = a09984f7d2ed4785
  A[ 148] = b3d375606156d954
  A[ 149] = 8f7d6fb5316a6189
  A[ 150] = 1b87a1d5504f7fc3
  A[ 151] = e3d53e19846c0868
  A[ 152] = 9dfbc0507d476a7d
  Output (16 words of chaining values):
  A[ 153] = 2d1abe0601b2e6b0 output chaining
value C[0]
  A[ 154] = 61d59fd2b7310353 output chaining
value C[1]
  A[ 155] = ea7da28dec708ec7 output chaining
value C[2]
  A[ 156] = a63a99a574e40155 output chaining
value C[3]
  A[ 157] = 290b4fabe80104c4 output chaining
value C[4]
  A[ 158] = 8c6a3503cf881a99 output chaining
value C[5]
  A[ 159] = e370e23d1b700cc5 output chaining
value C[6]
```

```
  A[  160]  =  4492e78e3fe42f13  output  chaining
value C[7]
  A[  161]  =  df6c91b7eaf3f088  output  chaining
value C[8]
  A[  162]  =  aab3e19a8f63b80a  output  chaining
value C[9]
  A[  163]  =  d987bdcbda2e934f  output  chaining
value C[10]
  A[  164]  =  aeae805de12b0d24  output  chaining
value C[11]
  A[  165]  =  8854c14dc284f840  output  chaining
value C[12]
  A[  166]  =  ed71ad7ba542855c  output  chaining
value C[13]
  A[  167]  =  e189633e48c797a5  output  chaining
value C[14]
  A[  168]  =  5121a746be48cec8  output  chaining
value C[15]

  8854c14dc284f840ed71ad7ba542855ce189633e48c797
a55121a746be48cec8
```

Dari hasil di atas, dapat dilihat bahwa hasil hash dari string “abc” dengan MD6 adalah :

```
  8854c14dc284f840ed71ad7ba542855ce189633e48c797
a55121a746be48cec8
```

D. Analisis Perbandingan Hasil

Analisis Algoritma MD5 :

1. Panjang message digest yang dihasilkan MD5 memiliki panjang pesan sebesar 128 bit. Hal ini membuat algoritma MD5 adalah termasuk algoritma yang paling lemah dibandingkan dengan MD6 dan WHIRLPOOL yang memiliki panjang pesan 512 bit.
2. Sudah ditemukan *collision* dalam pengimplementasian MD5
3. Komputasi yang dilakukan MD5 lebih cepat dibandingkan dengan MD6 dan WHIRLPOOL. Hal ini dapat dilihat dari hasil ujicoba pada sub-bab sebelumnya, yaitu denga data test yag sama, algoritma MD5 menghasilkan message digest dalam waktu 9 millisecond, sedangkan WHIRLPOOL membutuhkan waktu 24 millisecond.

Analisis Algoritma WHIRLPOOL :

1. Algoritma WHIRLPOOL mendukung eksekusi paralel
2. Algoritma WHIRLPOOL tidak membutuhkan ruang penyimpanan yang besar (baik uutk kode maupun utuk tabel)
3. Panjang message digest yang dihasilkan WHIRLPOOL memilki panjang pesan sebesar 512 bit, sehingga meningkatka proteksi terhadap serangan
4. Fungsi hash WHIRLPOOL belum dideteksi adanya *collision*.

Analisis Algoritma MD6

1. Algoritma MD6 dapat memanfaatkan banyak core dalam pemrosesan dimana perhitungan dapat dilakukan secara paralel
2. MD6 memiliki ukuran blok input besar yang mampu meningkatkan kemudahan penambahan input tambahan, efisiensi, paralelisme dan sekuriti.
3. Panjang message digest yang dihasilkan MD6 memilki panjang pesan sebesar 512 bit, sehingga meningkatkan proteksi terhadap serangan
4. Algoritma MD6 juga masih belum ditemukan collision dalam pengimplemetasiannya.

V. KESIMPULAN DAN SARAN

Dari hasil implementasi yang telah dilakukan, maka dapat disimpulkan bahwa kompleksitas atau tingkat kerumitan suatu algoritma pada fungsi hash sebanding atau berbanding lurus dengan tingkat keamanan fungsi hash tersebut.

Kesimpulan yang lain yang didapatkan adalah kompleksitas atau tingkat kerumitan suatu algoritma pada fungsi hash berbanding terbalik dengan kecepatan dari algoritma fungsi hash tersebut.

Dari perbandingan terhadap MD5, MD6, dan WHIRLPOOL, ketiga algoritma tersebut memiliki kelebihan dan kekurangan masing – masing. Pemilihan algoritma hash yang cocok dapat disesuaikan dengan kebutuhan.

REFERENCES

[1] http://groups.csail.mit.edu/cis/md6/
Diakses pada : 8 Mei 2011

[2] http://userpages.umbc.edu/~mabzug1/cs/md5/md5.html
Diakses pada : 8 Mei 2011

[3] http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html
Diakses pada : 8 Mei 2011

PERNYATAAN

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

Bandung, 9 Mei 2011

Roy Indra Haryanto
13508026