Perkembangan Enkripsi Fungsi Hash pada SHA (Secure Hash Algorithm)

Muharram Huda W.

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
e-mail: if18033@students.if.itb.acid

ABSTRAK

Verifikasi dan autentikasi merupakan hal yang yang penting dalam pengiriman informasi. Hal ini dikarenakan dalam pertukaran informasi, dibutuhkan kepercayaan terhadap keaslian dan keutuhan data. Fungsi hash merupakan salah satu fungsi yang memberika layanan untuk ferivikasi autentikasi karena fungsi ini menghasilkan nilai yang unik untuk setiap input. Fungsi hash ini juga disebut fungsi satu arah karena sangat sulit untuk mengembalikan ke input awal dengan fungsi hash. Fungsi hash ini suatu implementasi dari kriptografi. Tetapi bagaimanapun, hash juga mempunyai kelemahan. Makalah ini akan menjelaskan tentang perkembangan hash yang telah dianggap "paling" aman untuk kebutuhan pertukaran informasi.

Kata kunci: Hash, SHA, Kriptografi.

1. PENDAHULUAN

Hash adalah suatu teknik "klasik" dalam ilmu Komputer yang banyak digunakan dalam praktek enkripsi. *Hash* merupakan suatu metode yang secara langsung mengakses record-record dalam suatu tabel dengan melakukan transformasi aritmatik pada suatu input dari user yang biasanya merupakan bentuk string.

Fungsi Hash adalah suatu fungsi yang digunakan untuk mengenkripsi sebuah data menjadi data yang lebih kecil yang mengandung nilai unik yang merepresentasikan data sebelumnya. Nilai hash dari suatu fungsi hash akan memiliki panjang bit yang tetap untuk masukan apapun dengan panjang bit berapapun. Idealnya dari kriptografi fungsi hash, mempunya 3 keunggulan utama,

1. Mudah dan cepat menghasilkan nilai hash dari input apapun.
2. Sangat sulit untuk mengembalikan nilai hash ke input asli.
3. Sangat sulit untuk mengubah data asli tanpa mengubah hash.
4. Tidak ada input yang berbeda yang mempunyai nilai hash yang sama.

Kriptografi fungsi hash mempunyai banyak kegunaan dalam keamanan informasi, seperti pada tanda tangan digital, MACs (*Message authentication codes*), dan beberapa bentuk authetikasi lainnya.

2. Kriptografi, Fungsi Hash Satu Arah, dan Serangan Terhadap Hasil fungsi

Bagian ini menerangkan tentang kriptografi secara umum yaitu mengenai enkripsi dan dekripsi. Dan juga menjelaskan fungsi hash satu arah.

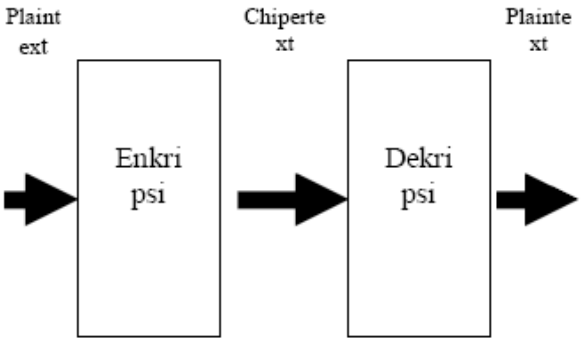
2.1 Kriptografi

Kriptografi secara umum adalah ilmu yang menjaga kerahasiaan suatu berita atau pesan. Selain itu kriptografi dapat juga diartikan sebagai ilmu yang mempelajari teknik-teknik matematika yang berhubungan dengan aspek keamanan informasi.

Kriptografi mempunyai 2 bagian penting, yaitu enkripsi dan dekripsi. Dasar matematis yang mendasari proses enkripsi dan dekripsi adalah relasi antara dua himpunan yaitu yang berisi elemen teks terang(*plainteks*) dan yang berisi sandi (chiperteks). Enkripsi dan dekripsi merupakan fungsi transformasi antara himpunan-himpunan tersebut. Apabila elemen-elemen teks terang dinotasikan dengan P, elemen-elemen teks sandi dinotasikan dengan C. sedang untuk proses enkripsi dinotasikan dengan E dan dekripsi dinotasikan dengan D.

Enkripsi : E(P)=C (1)
Dekripsi : D(C)=P atau D(E(P))=P (2)
Ref[1]

Pada gambar dibawah ini dapat dilihat bahwa masukan berupa *plaintext* akan masuk ke dalam blok enkripsi dan keluarannya berupa *chipertext*, kemudian *chipertext* akan masuk ke dalam blok dekripsi dan keluarannya akan kembali menjadi *plaintext* semula.



Gambar 2.1. analogi dalam kriptografi

2.2 Fungsi Hash Satu Arah

Fungsi *hash* satu arah (o*ne-way hash function*) adalah *hash function* yangbekerja satu arah, yaitu suatu *hash function* yang dengan mudah dapat menghitung *hash value* dari *pre-image*, tetapi sangat sukar untuk menghitung *pre-image* dari *hash value*.

Sebuah fungsi *hash* satu arah, *H(M)*, beroperasi pada suatu *pre-image* pesan *M* dengan panjang sembarang, dan mengembalikan nilai *hash h* yang memiliki panjang tetap. Dalam notasi matematika fungsi *hash* satu arah dapat ditulis sebagai:

$$H = H(M), \text{ dengan h memiliki panjang b} \quad (3)$$

Ada banyak fungsi yang mampu menerima input dengan panjang sembarang dan menghasilkan output dengan panjang tetap, tetapi fungsi *hash* satu arah memiliki karakteristik tambahan yang membuatnya satu arah :

- Diberikan M, mudah menghitung h.
- Diberikan h, sulit menghitung M agar H(M) = h.
- Diberikan M, sulit menemukan pesan lain, M', agar H(M) = H(M').

Dalam dunia nyata, fungsi *hash* satu arah dikembangkan berdasarkan ide sebuah fungsi kompresi. Fungsi satu arah ini menghasilkan nilai *hash* berukuran n bila diberikan input berukuran b. Input untuk fungsi kompresi adalah suatu blok pesan dan hasil blok teks sebelumnya. Sehingga *hash* suatu blok M, adalah:

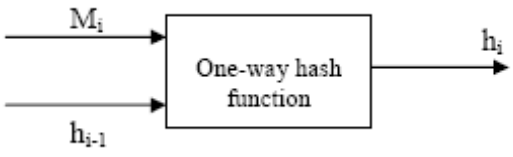
$$h_i = f(M_i, h_{i-1}) \quad (4)$$

Dengan

h_i = nilai hash saat ini

M_i = blok pesan saat ini

h_{i-1} = nilai hash blok teks sebelumnya



Gambar 2.2. analogi dalam fungsi satu arah

2.3 Serangan Terhadap Hasil fungsi

Serangan terhadap kriptogafi fungsi hash sejauh ini ada 3, yaitu:

1. *First Preimage attack*
   Ini adalah serangan untuk mencari nilai input dari nilai hasil hash.
2. *Second Preimage Attack*
   Ini adalah serangan untuk mencari nilai input yang berbeda dari 1 nilai hash
3. *Collision Attack or Birthday Attack*
   Ini adalah serangan dengan menemukan 2 input yang berbeda yang mempunyai nilai hash yang sama.

3. Perkembangan SHA

Bagian ini menerangkan tentang SHA dari masa ke-masa dan juga menjelaskan algoritma hash yang juga terbilang aman selain SHA.

3.1 SHA-0

SHA-0 , dikatakan mempunyai kekuatan 2^{80}, maksudnya adalah, dengan melakukan *brute force*, data yang disandikan dengan algoritma ini akan pecah, dalam hal ini mengalami *collision* yaitu kesamaan nilai hash dari input yang berbeda.

SHA-0 mengalami beberapa kelemahan dalam *Collition Attack* hal ini disertai fakta bahwa:

1. Pada saat konferensi CRYPTO 98, dua orang peneliti dari Perancis mempresentasikan sebuah attack terhadap SHA 0 dimana collisions dapat ditemukan dengan kompleksitas 2^{61}. Nilai kompleksitas tersebut lebih rendah dari nilai kompleksitas ideal suatu hash function, yaitu 2^{80}.
2. Pada tahun 2004, Chen dan Biham telah berhasil menemukan near-collisions untuk SHA 0, yaitu menemukan dua berita yang hampir mempunyai nilai hash yang sama dimana dari 160 bit output yang dihasilkan, ternyata 142 bitnya sama. Mereka juga menemukan full collisions pada SHA 0 dengan 62 round dari total 80 round.
3. Pada 12 Agustus 2004, sebuah collision untuk full SHA 0 diumumkan oleh empat orang peneliti yaitu Joux, Carribault, Lemuet, dan Jalby. Dengan menggunakan Chabaud dan Joux attack, ternyata ditemukan collision dengan kompleksitas 251 dan membutuhkan 80.000 jam dengan menggunakan superkomputer yang di dalamnya terdapat 256 buah prosesor Itanium 2.
4. Pada 17 Agustus 2004, saat Rump Session CRYPTO 2004, sebuah hasil pendahuluan diumumkan oleh Wang, Feng, Lai, dan Yu tentang attack terhadap SHA 0, MD 5, dan fungsi hash lainnya. Kompleksitas attack mereka terhadap SHA 0 adalah sekitar 240, jauh lebih baik dibandingkan attack yang dilakuka oleh Joux dan lainnya.

5. Pada Februari 2005, Wang, Yiqun Lisa Yin, dan Yu kembali melakukan sebuah attack. Mereka menemukan collision pada SHA 0 dalam 239 operasi.

3.2 SHA-1

SHA-1 mempunyai kapasitas input message 2^{64}-1, dengan hasil hash 160 bits dan evaluasi kekuatan hash 2^{80}

3.2.1 Algoritma SHA-1

Misal SHA-1 digunakan untuk meng-hash sebuah pesan, M, yang mempunyai panjang maksimum 2^{64}-1 bits. Algoritma ini menggunakan urutan dari 80 kali 32-bit kata, dengan menggunakan 5 variabel yang menampung 32 bits per variabel, dan hasil hashnya. Jadi, hasilnya adalah 160 bit hasil hash.

Misal urutan kata itu diberi tanda $W_0,W_1,\ldots,W_{79}$. 5 variabel itu diberi tanda A,B,C,D,E. kata hasil hash diberi tanda H_0,H_1, H_2,H_3, H_4.

Lalu masuk ke langkah:

1. Melakukan padding terhadap pesan sehingga panjangnya adalah 448 modulus 512. Sehingga menjadi M=448+N.512 . lalu membagi M menjadi $M^{(1)}$, $M^{(2)}$, ….$M^{(N)}$.
2. 64 bit sisanya adalah representasi biner dari panjang pesan.
3. Lalu masuk ke komputasi Hash, dengan menggunakan iterasi $M^{(i)}$ dengan i=1 sampai N dengan langkah:

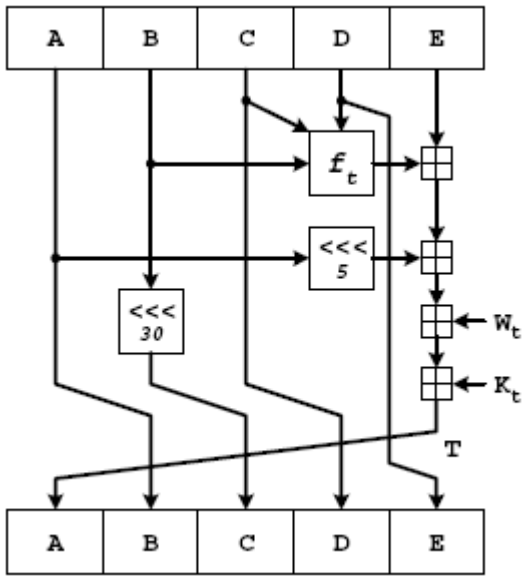
- Gunakan iterasi dari W, dengan simbol W_t :

$$W_t = \begin{cases} M_t^{(i)} & , 0 \le t \le 15 \\ \text{ROTL}^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) & , 16 \le t \le 79 \end{cases} \quad (4)$$

- Menginisialisasi A,B,C,D,E dengan hasil dari hash sebelumnya
- Lalu melakukan proses terhadap A,B,C,D,E sesuai rumus diatas:

$$\begin{aligned} T &= \text{ROTL}^5(A) + f_t(B,C,D) + E + K_t + W_t \\ E &= D \\ D &= C \\ C &= \text{ROTL}^{30}(B) \\ B &= A \\ A &= T \end{aligned} \quad (5)$$

- Lalu menambahkan hasil A,B,C,D,E dengan hash sebelumnya sesuai urutan, lalu memasukkannya ke hasil hash yang sekarang juga sesuai urutan.



Gambar 3.1. analogi dalam komputasi hash pada SHA-1

3.2.2 Code SHA-1

Note 1: All variables are unsigned 32 bits and wrap modulo 2^{32} when calculating
Note 2: All constants in this pseudo code are in big endian.
Within each word, the most significant byte is stored in the leftmost byte position

Initialize variables:
h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
h4 = 0xC3D2E1F0

Pre-processing:
append the bit '1' to the message
append 0 ≤ k < 512 bits '0', so that the resulting message length (in *bits*)
is congruent to 448 ≡ −64 (mod 512)
append length of message (before pre-processing), in *bits*, as 64-bit big-endian integer

Process the message in successive 512-bit chunks:
break message into 512-bit chunks
for each chunk
break chunk into sixteen 32-bit big-endian words w[i], 0 ≤ i ≤ 15

Extend the sixteen 32-bit words into eighty 32-bit words:
for i **from** 16 to 79
w[i] = (w[i-3] **xor** w[i-8] **xor** w[i-14] **xor** w[i-16]) **leftrotate** 1

```
    Initialize hash value for this chunk:
    a = h0
    b = h1
    c = h2
    d = h3
    e = h4

    Main loop:
    for i from 0 to 79
        if 0 ≤ i ≤ 19 then
            f = (b and c) or ((not b) and d)
            k = 0x5A827999
        else if 20 ≤ i ≤ 39
            f = b xor c xor d
            k = 0x6ED9EBA1
        else if 40 ≤ i ≤ 59
            f = (b and c) or (b and d) or (c and d)
            k = 0x8F1BBCDC
        else if 60 ≤ i ≤ 79
            f = b xor c xor d
            k = 0xCA62C1D6

        temp = (a leftrotate 5) + f + e + k + w[i]
        e = d
        d = c
        c = b leftrotate 30
        b = a
        a = temp

    Add this chunk's hash to result so far:
    h0 = h0 + a
    h1 = h1 + b
    h2 = h2 + c
    h3 = h3 + d
    h4 = h4 + e

Produce the final hash value (big-endian):
digest = hash = h0 append h1 append h2 append h3
append h4
```

3.2.3 Kriptoanalisis SHA-1

Banyak para ahli kriptografi mengadakan penelitian terhadap kekuatan kriptosistem SHA 1. Setelah berjalan kurang lebih 10 tahun, akhirnya apada tahun 2005, Rijmen dan Oswald mempublikasikan serangan pada versi SHA 1 yangdireduksi (hanya menggunakan 53 round dari 80 round) dan hasilnya telah ditemukan collision dengan kompleksitas sekitar 2^{80} operasi. Pada tahun yang sama, Xiayoun Wang, Yiqun Lisa Yin, dan Hongbo Yo juga mengumumkan serangan yang dapat menemukan collision pada versi penuh SHA 1 yang memerlukan sekitar 2^{69} operasi.

3.3 SHA-2

SHA-2 tidak begitu banyak digunakan walaupun untuk keamanan, SHA-2 termasuk salah satu yang paling aman dantara algoritma yang lain. hal ini dikarenakan proses yang berlarut2 yang menyebabkan lamanya waktu dalam melakukan hash.

SHA-2 sendiri dibagi 4 type, tergantung dari bit keluaran hasil hash-nya, yaitu SHA-224,SHA-256,SHA-384,SHA-512.

Selain lamanya saal melakukan hashing, algoritma SHA-2 tidak support digunakan pada windows XP SP2 dan sebelumnya.

3.2.2 Code SHA-256

```
Note 2: All constants in this pseudo
code are in big endian

Initialize variables
(first 32 bits of the fractional parts
of the square roots of the first 8
primes 2..19):
h0 := 0x6a09e667
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
h6 := 0x1f83d9ab
h7 := 0x5be0cd19

Initialize table of round constants
(first 32 bits of the fractional parts
of the cube roots of the first 64
primes 2..311):
k[0..63] :=
   0x428a2f98, 0x71374491, 0xb5c0fbcf,
0xe9b5dba5, 0x3956c25b, 0x59f111f1,
0x923f82a4, 0xab1c5ed5,
   0xd807aa98, 0x12835b01, 0x243185be,
0x550c7dc3, 0x72be5d74, 0x80deb1fe,
0x9bdc06a7, 0xc19bf174,
   0xe49b69c1, 0xefbe4786, 0x0fc19dc6,
0x240ca1cc, 0x2de92c6f, 0x4a7484aa,
0x5cb0a9dc, 0x76f988da,
   0x983e5152, 0xa831c66d, 0xb00327c8,
0xbf597fc7, 0xc6e00bf3, 0xd5a79147,
0x06ca6351, 0x14292967,
   0x27b70a85, 0x2e1b2138, 0x4d2c6dfc,
0x53380d13, 0x650a7354, 0x766a0abb,
0x81c2c92e, 0x92722c85,
   0xa2bfe8a1, 0xa81a664b, 0xc24b8b70,
0xc76c51a3, 0xd192e819, 0xd6990624,
0xf40e3585, 0x106aa070,
   0x19a4c116, 0x1e376c08, 0x2748774c,
0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a,
0x5b9cca4f, 0x682e6ff3,
   0x748f82ee, 0x78a5636f, 0x84c87814,
0x8cc70208, 0x90befffa, 0xa4506ceb,
0xbef9a3f7, 0xc67178f2
```

```
Pre-processing:
append the bit '1' to the message
append k bits '0', where k is the
minimum number >= 0 such that the
resulting message
    length (in bits) is congruent to
448 (mod 512)
append length of message (before pre-
processing), in bits, as 64-bit big-
endian integer

Process the message in successive 512-
bit chunks:
break message into 512-bit chunks
for each chunk
    break chunk into sixteen 32-bit
big-endian words w[0..15]

    Extend the sixteen 32-bit words
into sixty-four 32-bit words:
    for i from 16 to 63
        s0 := (w[i-15] rightrotate 7)
xor (w[i-15] rightrotate 18) xor (w[i-
15] rightshift 3)
        s1 := (w[i-2] rightrotate 17)
xor (w[i-2] rightrotate 19) xor (w[i-2]
rightshift 10)
        w[i] := w[i-16] + s0 + w[i-7] +
s1

    Initialize hash value for this
chunk:
    a := h0
    b := h1
    c := h2
    d := h3
    e := h4
    f := h5
    g := h6
    h := h7

    Main loop:
    for i from 0 to 63
        s0 := (a rightrotate 2) xor (a
rightrotate 13) xor (a rightrotate 22)
        maj := (a and b) xor (a and c)
xor (b and c)
        t2 := s0 + maj
        s1 := (e rightrotate 6) xor (e
rightrotate 11) xor (e rightrotate 25)
        ch := (e and f) xor ((not e)
and g)
        t1 := h + s1 + ch + k[i] + w[i]

        h := g
        g := f
        f := e
        e := d + t1
        d := c
        c := b
        b := a
        a := t1 + t2

    Add this chunk's hash to result so
far:
    h0 := h0 + a
    h1 := h1 + b
    h2 := h2 + c
    h3 := h3 + d
    h4 := h4 + e
    h5 := h5 + f
    h6 := h6 + g
    h7 := h7 + h

Produce the final hash value (big-
endian):
digest = hash = h0 append h1 append h2
append h3 append h4 append h5 append h6
append h7
```

3.4 Dynamic SHA-1 dan 2

Dynamic SHA 1 dan 2 adalah kandidat untuk menjadi pengganti bagi SHA2, karena kecepatan yang lebih besar dari SHA-2. Tapi menurut penelitian, dengan serangan collision Dynamic SHA-1 hanya bertahan di 2^{22} sedangkan SHA-2-256 bertahan di 2^{52}, sebuah angka yang rendah untuk kekuatan sebuah hash, sehingga kedua algoritma ini dianggap masih belum bisa menjadi pengganti bagi SHA-2.

Berikut hasil tiap-tiap serangan terhadap kedua algoritma.

Tabel 3.1 Tabel hasil penyerangan terhadap macam-macam Dynamic SHA

Hash Function	Attack	Complexity	Section
Dynamic SHA-256	Collision	2^{21}	3
Dynamic SHA-512	Collision	2^{22}	3,C
Dynamic SHA-256	Second preimage	2^{216}	4
Dynamic SHA-512	Second preimage	2^{256}	4,C
Dynamic SHA-256	First preimage	2^{225}	4
Dynamic SHA-512	First preimage	2^{262}	4,C
Dynamic SHA2-256	Collision	2^{52}	5
Dynamic SHA2-512	Collision	2^{85}	5,C

3.5 Tiger

Fungsi kompresi Tiger berdasarkan pada penerapan fungsi internal "*block chipper*", yang memerlukan 192 bit plaintext dan 512 bit *key* untuk menghitung 192 bit "*ciphertext*". Fungsi "*block chiper*" diterapkan menurut konstruksi Davies-Meyer : 512 bit blok *message* digunakan sebagai kunci (*key*) untuk meng-encrypt 192 bit *hash value*, dan kemudian input *hash value* di-

feedforward untuk menjadikan seluruh fungsi *non-invertible*.

Tiger dirancang dengan arsitektur 64 bit. Untuk itu, akan dinotasikan sebuah *unsigned integer* 64 bit sebagai sebuah "*word*". *Word* akan direpresentasikan dalam bentuk heksadesimal. Tiger menggunakan operasi-operasi aritmetika, bit-wise XOR, NOT, operasi logika shift, dan aplikasi S-Box. Operasi aritmetika terhadap word adalah modulus 264. *Hash value* direpresentasikan secara internal sebagai 3 buah register 64 bit a, b, dan c. Register-register ini diinisialisasi dengan *h0* dimana :

a = 0x0123456789ABCDEF
b = 0xFEDCBA9876543210
c = 0xF096A5B4C3B2E187

Masing-masing suksesor 512 bit blok message dibagi kedalam delapan word 64 bit x0, x1, … , x7, dan komputasi berikutnya dilakukan untuk meng-update *hi* hingga *hi+1*.

Komputasi ini terdiri dari 3 laluan (passing), dan di antara setiap laluan tersebut terdapat sebuah *key schedule*, yaitu suatu tranformasi invertible dari data input yang mencegah penyerang memaksakan input di ketiga round tersebut. Pada akhirnya terdapat suatu tahap *feedforward* yang mana nilai baru *a, b* dan *c* dikombinasikan dengan nilai awalnya untuk menghasilkan *hi+1*.

```
save_abc
pass(a,b,c,5)
key_schedule
pass(c,a,b,7)
key_schedule
pass(b,c,a,9)
feedforward
```

dengan

`save_abc` menyimpan nilai *hi*

```
aa = a ;
bb = b ;
cc = c ;
```

`pass(a,b,c,mul)` adalah

```
round(a,b,c,x0,mul);
round(b,c,a,x1,mul);
round(c,a,b,x2,mul);
round(a,b,c,x3,mul);
round(b,c,a,x4,mul);
round(c,a,b,x5,mul);
round(a,b,c,x6,mul);
round(b,c,a,x7,mul);
```

dengan `round(a,b,c,x,mul)` adalah

```
c ^= x ;
a -= t1[c_0] ^ t2[c_2] ^ t3[c_4] ^
t4[c_6] ;
b += t4[c_1] ^ t3[c_3] ^ t2[c_5] ^
t1[c_7] ;
b *= mul;
```

dan c_i adalah byte ke-i dari c ($0 \leq i \leq 7$). Catatan, kita menggunakan notasi pemrograman dengan bahasa C, dengan ^ melambangkan operator XOR, dan notasi X op=Y berati X = X op Y untuk setiap operator op.

`key_schedule` adalah

```
x0 -= x7 ^ 0xA5A5A5A5A5A5A5A5;
x1 ^= x0;
x2 += x1;
x3 -= x2 ^ ((~x1)<<19);
x4 ^= x3;
x5 += x4;
x6 -= x5 ^ ((~x4)>>23);
x7 ^= x6;
x0 += x7;
x1 -= x0 ^ ((~x7)<<19);
x2 ^= x1;
x3 += x2;
x4 -= x3 ^ ((~x2)>>23);
x5 ^= x4;
x6 += x5;
x7 -= x6 ^ 0x0123456789ABCDEF;
```

dengan << dan >> adalah operator logika *shift left* dan *shift right*.

`feedforward` adalah

```
a ^= aa ;
b -= bb ;
c += cc ;
```

Resultan register-register a, b, c adalah 192 bit untuk hash value hi+1. x1, x2, ... , x7 pada passing (laluan) kedua dan ketiga. Akhirnya, nilai akhir *hn* diambil sebagai output Tiger/192.

3.5.1 Keunggulan Tiger

Beberapa aspek keamanan yang dapat dikemukakan sehubungan dengan fungsi hash Tiger adalah :

1. Ketidak-linear-an sebagian besar berasal dari S-Box 8 bit hingga 64 bit. Ini lebih baik dibanding hanya mengkombinasikan penjumlahan dan XOR, dan ini cenderung pada seluruh bit-bit output, bukan hanya pada bit-bit tetangga.
2. Terdapat *avalanche* yang kuat, pada setiap bit *message* di ketiga register setelah putaran *(round*) ketiga. Ini lebih cepat dibanding fungsi hash yang lain. *Avalanche* pada *word* 64 bit (dan 64 bit S-Box) lebih cepat dibanding ketika menggunakan *word-word* yang lebih pendek.
3. Seperti yang telah dijelaskan sebelumnya, semua serangan pada MD / Snefru menargetkan pada satu blok *intermediate*. Penambahan nilai *intermediate* ke 192 bit membantu menghalangi serangan tersebut.
4. *Key schedule* menjamin bahwa perubahan sejumlah kecil bit *message* mempengaruhi banyak bit selama proses *passing* (laluan) berlangsung. Bersama dengan avalanche yang kuat, ia dapat membantu Tiger untuk dapat bertahan terhadap serangan *Dobbertin's differential attack* yang terjadi pada MD4 (dimana perubahan bit-bit message tertentu mempengaruhi paling banyak 2 bit di banyak putaran (*round*), dan kemudian perbedaan kecil ini dapat dikeluarkan di laluan (*passing*) terakhir).
5. Perkalian register b di tiap putaran (*round*) juga memberikan ketahanan terhadap serangan, dikarenakan

jaminan bahwa bit-bit yang digunakan sebagai input di S-Box pada putaran sebelumnya dicampur ke dalam SBox lainnya juga, dan dalam S-Box yang sama dengan sebuah perbedaan input. Perkalian ini juga mencegah serangan *related-key* pada fungsi hash dikarenakan perbedaan konstanta di setiap putaran.

6. *Feedforward* melindungi terhadap *birthday atack* yang terjadi di pertengahan yang menemukan *preimage* fungsi hash (meskipun kompleksitasnya mencapai 296).

4. Aplikasi SHA

SHA-1 adalah yang paling banyak digunakan publik daripada SHA-SHA lain. SHA banyak digunakan pada aplikasi keamanan dan protokol, seperti:

a. Transport Layer Security dan Secure Socket Layer
   Adalah protokol kriptografi yang menyediakan keamanan untuk komunikasi antar network seperti internet.
b. Pretty Good Privacy (PGP)
   Adalah program komputer yang menyediakan privasi kriptografi dan autentikasi. Biasanya digunakan untuk sign in, enkripsi dan dekripsi e-mail, dan menambah kemanan pengiriman e-mail.
c. Secure Shell (SSH)
   Protokol network yang menjadi jalur keamanan antara 2 alat network untuk bertukar informasi, biasanya digunakan di Linux dan Unix.
d. S/MIME(Secure/Multi purpose Internet Mail Extensions)
e. Internet Protocol Security(IPsec)
   Program pengamanan komunikasi Internet Protocol dengan mengautentikasi dan mengenkripsi setiap paket dan data stream.

IV. KESIMPULAN

Fungsi Hash adalah suatu fungsi untuk mengenkripsi suatu data. Fungsi hash memiliki algoritma yang berbeda-beda sesuai perkembangan jaman yang menuntut untuk terus menjadi lebih aman dan cepat.

SHA-0,SHA-1,SHA-2 dirasa masih kurang dalam mengenkripsi data, sehingga dibutuhkan SHA yang lebih mangkus dalam mengamankan data.

Ada beberapa yang bisa memperkuat enkripsi untuk menggantikan SHA-2, beberapa diantaranya adalah Dynamic SHA-1 dan 2 dan Tiger.

1. Dynamic SHA-1 dan 2 lebih cepat dalam proses, tetapi kurang bisa mengamankan data
2. Tiger terbukti bisa tahan terhadap beberapa serangan, tapi karena ada perkalian pada setiap register b membuat tiger lebih lambat daripada dynamic SHA-1 dan 2

REFERENSI

http://id.wikipedia.org/wiki/Kriptografi, waktu akses 20 desember 2009

http://en.wikipedia.org/wiki/Hash_function, waktu akses 20 desember 2009

http://en.wikipedia.org/wiki/Cryptographic_hash_function, waktu akses 20 desember 2009

http://en.wikipedia.org/wiki/SHA_hash_functions, waktu akses 20 desember 2009

http://en.wikipedia.org/wiki/Transport_Layer_Security, waktu akses 20 desember 2009

http://www.wisdom.weizmann.ac.il/~orrd/crypt/DSHA2.pdf, waktu akses 20 desember 2009

http://www.itl.nist.gov/fipspubs/fip180-1.htm