Modifikasi Fungsi Hash Berdasarkan Algoritma RC4

Faradina Ardiyana
Program Studi Teknik Informatika Institut Teknologi Bandung, Jl.Ganesha 10 Bandung 40132
Email: if14067@students.if.itb.ac.id

Abstract *– pada makalah ini, saya akan mengusulkan fungsi hash baru yang berdasar pada RC4 dan kemudiannya akan disebut sebagai RC4-Hash. Fungsi hash H ini menghasilkan nilai(h) dengan panjang berubah-ubah dari 16 bytes hingga 64 bytes. Fungsi RC4-Hash memiliki beberapa keuntungan dibandingkan dengan fungsi hash yang lainnya. Ketepatgunaannya sebanding dengan fungsi hash lainnya. Dapat diketahui serangan baru-baru ini terhadap fungsi hash MD4, MD5, SHA-0, SHA-1 dan RIPEMD, oleh karena itu dibutuhkan strategi desain fungsi hash baru. Saya akan mengusulkan desain fungsi hash dengan struktur internal baru. Analisis keamanan dari RC4-Hash dapat diperoleh dengan cara yang sama dengan analisis keamanan terhadap RC4 juga serangan yang dilakukan terhadap fungsi hash berbeda. Kriteria desain fungsi hash baru ini berbeda dengan semua fungsi hash sebelumnya yang sudah diketahui. Diharapkan fungsi hash ini akan lebih terjamin keamanannya. Pada makalah ini juga akan diberikan deskripsi dari fungsi hash dan RC4.*

Kata Kunci*: fungsi hash, RC4, Collision Attack, Preimage Attack.*

1.PENDAHULUAN

Fungsi hash merupakan kepentingan pokok dari protokol kriptografi. Fungsi hash adalah fungsi yang secara efisien mengubah string input dengan panjang berhingga menjadi string output dengan panjang tetap yang disebut nilai hash. Kita semua tahu bahwa tanda-tangan digital sangatlah penting dalam keamanan informasi. Keamanan dari tanda-tangan digital tergantung pada kekuatan kriptografi yang mendasari fungsi hash. Aplikasi lainnya dari fungsi hash dalam kriptografi adalah integritas data, *time stamping*, verifikasi password, watermark digital, *e-cash*, dan dalam banyak protokol kriptografi lainnya. Fungsi hash biasanya dibuat dari coretan(*scratch*) atau dibuat dari cipher blok dalam *black box manner*. Beberapa fungsi hash yang sudah diketahui yang dibangun dari sebuah coretan adalah SHA-*family*, MD4, MD5, RIPEMD, Tiger, HAVAL, dan lain-lain. Sedangkan fungsi hash PGV, MDC2 dibuat dalam sebuah *black box manner*. Sejak diantara fungsi hash SHA-*family* SHA-0, SHA-1 sudah dapat dipatahkan, kita tidak dapat mempercayakan tentang keamanan dari algoritma lainnya yang ada pada keluarga SHA karena dasar pembuatan mereka sama. Demikian juga MD4, MD5, RIPEMD, dan HAVAL juga sudah dapat dipatahkan. Jadi kita butuh untuk membuat baru, algoritma keluaran hash yang berubah-ubah dengan struktur internal yang berbeda menjaga keamanan dan efisiensi.

2. RC4

RC4 (atau *ARCFOUR*) adalah cipher aliran yang digunakan secara luas pada sistem keamanan seperti protokol SSL (*Secure Socket Layer*). Algoritma kriptografi ini sederhana dan mudah diimplementasikan. RC4 membangkitkan aliran kunci yang kemudian di-*XOR*-kan dengan plainteks pada waktu enkripsi (atau di-*XOR*-kan dengan bit-bit cipherteks pada waktu dekripsi). RC4 memproses data dalam ukuran *byte*, 1 *byte* = 8 bit. Untuk membangkitkan aliran kunci, cipher menggunakan status internal yang terdiri dari dua bagian :

1. Permutasi angka 0 sampai 255 (256 kemungkinan *byte*) di dalam larik S_0, S_1, …, S_{255}. Permutasi merupakan fungsi dari kunci *U* dengan panjang variabel.
2. Dua buah 8-bit pencacah indeks, *i* dan *j*.

Demikian RC4 memiliki *internal state* yang sangat besar yaitu $\log_2(2^8! \times (2^8)^2) \approx 1700$ bit.

2.1 Algoritma RC4

Misalkan $[N] := [0, N-1] := \{0,1,\ldots,N-1\}$ dan Perm(*A*) adalah semua permutasi pada *A*. Pada makalah ini, akan diperhatikan pada **Perm**([*N*]) (atau ditulis sebagai **Perm**), dimana $N = 256 = 2^8$. Untuk S € **Perm**, berikan *S*[*i*] untuk nilai dari permutasi *S* pada posisi *i* € [*N*]. Modulo tambahan *N* diberikan dengan "+", sebaliknya akan dinyatakan kosong. Fungsi **Swap**(*S*[*i*],*S*[*j*]) adalah fungsi operasi pertukaran nilai dalam *S*[*i*] dan *S*[*j*]. Berikut ini adalah algoritma *key-scheduling* (*RC4-KSA*) dan algoritma pembangkitan kunci (*Pseudo-Random Byte Generation/RC4-PRBG* atau *PRBG*).

RC4-KSA(K)	RC4-PRBG(S)

for i = 0 to N-1
S[i] = I; j=0;
for i = 0 to N-1
j=j + S[i] + K[I mod *k*];
Swap(S[i],S[j]);
	i = 0, j = 0
	Pseudo-Random Bytes Geberation:
	i = (i+1) mod N;
	j = (j+S[i]) mod N;
	Swap(S[i],S[j]);
	out = S[(S[i]+S[j]) mod N];

Disini $K = K[0] \| \ldots \| K[_K - 1]$, $K[i] \in [N]$, K adalah aliran kunci dan $_K$ adalah ukuran dari kunci rahasia dalam byte.

2.2 Beberapa Analisis Keamanan Relevan dari RC4

Terdapat beberapa serangan terhadap RC4, dan selanjutnya akan dibahas karena berhubungan dengan analisis keamanan terhadap RC4-Hash.

2.2.1 Distribusi setelah Algoritma *Key-Scheduling* (*RC4-KSA*) Berakhir Seragam (sama)

RC4 dapat dilihat sebagai perkiraan akhir dari pertukaran kocokan (*exchange shuffle*). Dalam pertukaran kocokan, nilai j dalam algoritma *key-scheduling* dipilih secara acak (tidak seperti RC4-KSA dimana nilai tersebut di-*update* secara rekursif berdasarkan kunci rahasia). Walaupun RC4-KSA tidak sama dengan *exchange shuffle*, setidaknya ada yang berharap untuk kesamaan sifat. Untuk lebih jelasnya, anggap bahwa jika K dipilih secara acak, maka distribusi pasangan (S,j) setelah pelaksanaan RC4-KSA nilainya mendekati distribusi persamaan, (S, j) = RC4-KSA(K) adalah secara keseluruhan didistribusikan pada **Perm** × $[N]$ asalkan K dipilih secara keseluruhan.

2.2.2 Distribusi dari hasil keluaran RC4-PRBG tidak seragam (sama)

Terdapat banyak pengamatan yang membuktikan bahwa distribusi dari RC4-PRBG(S) tidak dapat disamakan bahkan jika kita mengasumsikan bahwa S secara keseluruhan didistribusikan. Sebagai contoh :

1. Mantin dan Shamir menunjukkan bahwa kemungkinan dari byte kedua menjadi kosong (nol) nilainya mendekati $2/N$ dibandingkan dengan nilai kemungkinan $1/N$ pada kasus pembangkitan acak byte.
2. Paul dan Preneel menunjukkan bahwa kemungkinan nilai dua byte pertama sama mendekati $1/N\ (1 - 1/N)$.
3. Finney menetapkan ditingkatan i manapun dalam RC4-PRBG adalah sebuah pasangan $(S, j) \in$ **Perm** × $[N]$ dimana $j = i + 1$ dan $S[j] = 1$. Yang dapat mengecek jika ada *finney state* dalam PRBG tepat sebelum melakukan update I maka *state* selanjutnya juga *Finney*, begitupun juga sebaliknya. *Finney state* hanya timbul dari *finney state* saja. Gampang dilihat jika $(S, 1)$ adalah *finney state* pada tingkatan $i = 0$, maka semua hasil keluaran N dari PRBG berbeda. Kemungkinan sebuah pasangan (S, j) dipilih secara acak utuk beberapa i merupakan sebuah *finney state* adalah $1/N^2$. inilah yang membuat hasil dari PRBG tidaklah seragam.

2.3 Fungsi Hash

Fungsi hash biasanya digambarkan sebagai berikut :

Pertama-tama dibentuk fungsi kompresi $C : \{0, 1\}^c \times \{0, 1\}^c \rightarrow \{0, 1\}^c$. Ditunjukkan $C(h, x) = h'$ dengan $h \xrightarrow[x]{} h'$. Kemudian diberikan pesan M sehingga $|M|<2^{64}$, sebuah blok (*pad*) ditambahkan pada akhir pesan. Contoh, $\overline{M}$:= pad(M) = $M \| 10^k \| \text{bin}_{64}(|\ M\ |)$, dimana $\text{bin}_{64}(x)$ adalah representasi biner 64 bit dari x dan k adalah bilangan bulat positif yang paling sedikit sedemikian $|\ M\ | + k + 65 \equiv 0 \bmod a$.

$\overline{M}=M_1\|\ldots\|M_t$ (untuk bebrapa $t > 0$) dimana $|M_i|=a$.

Pilih nilai inisialisasi **IV** := $h_0 \in \{0, 1\}^c$ kemudian hitung nilai hash.

$h_0 \xrightarrow{M_1} h_1 \xrightarrow{M_2} \ldots \xrightarrow{M_{t-1}} h_{t-1} \xrightarrow{M_t} h_t$

Dimana h_t adalah nilai final hash, $H(M) = h_t$ dan $|h_i\ | = c$. Fungsi C adalah fungsi kompresi dan metode iterasi dikenal sebagai iterasi klasikal. Terdapat tiga dugaan penting dari keamanan fungsi hash :

1. Serangan *Collision* : cari $M_1 \neq M_2$, sehingga $H(M_1) = H(M_2)$.
2. Serangan Preimej : diberikan nilai acak $y \in \{0, 1\}^c$, cari nilai M sehingga $H(M) = y$.
3. Serangan Preimej kedua : diberikan pesan M_1, cari M_2 sehingga $H(M_1) = H(M_2)$.

Jika susah untuk menemukan dari serangan diatas maka dapat dikatakan fungsi hash sudah tahan terhadap serangan-serangan tersebut. Untuk sebuah fungsi hash c-bit, memerlukan pencarian yang mendalam sebesar $2^{c/2}$ kerumitan untuk *colision* dan 2^c kerumitan untuk kedua preimej dan preimej kedua. Dalam kasus serangan *colision*, serangan *birthday* terkenal menggunakan pencarian mendalam (*exhaustive search*). klasikal dengan kerumitan kurang dari 2^c. Kemudian, terdapat bentukan hash yang lebih luas. Pada gambaran ini, terdapat fungsi pokok (yang mendasari)

$C : \{0, 1\}^w \times \{0, 1\}^a \rightarrow \{0, 1\}^w$, disebut fungsi *compression-like* dan fungsi *post processing* $g : \{0, 1\}^w \rightarrow \{0, 1\}^c$. diberikan pesan lapisan (*padded*) $M = M_1 \| \ldots \| M_t$, dengan $|M_i\ | = a$, nilai hash dihitung sebagai berikut :

$h_0 \xrightarrow{M_1} h_1 \xrightarrow{M_2} \ldots \xrightarrow{M_{t-1}} h_{t-1} \xrightarrow{M_t} h_t$, $H(M) = g(h_t)$.

Jika w (ukuran keadaan menengah) sangat besar dibandingkan dengan c (ukuran hash akhir), maka keamanan dari H dapat diasumsikan kuat walaupun terdapat beberapa kelemahan dalam fungsi *compression-like* C. Serangan Kelsey-Schneier preimej kedua juga tidak akan bekerja jika nilai $w > 2c$. *Post processor* g tidak butuh terlalu cepat seperti sudah diaplikasikan sekali untuk setiap pesan. Kemudian, model fungsi hash yang lebih luas ini memiliki beberapa keuntungan daripada model fungsi hash lainnya seperti fungsi hash klasikal.

3. Algoritma RC4-Hash

Berikut akan saya jelaskan fungsi hash baru berdasarkan pada RC4, yang disebut RC4-Hash. Fungsi hash ini memiliki properti sebagai berikut :

1. Keluarga hash ditunjukkan sebagai RCH_ℓ, $16 \le l \le 64$ dimana RCH_ℓ: $\{0, 1\}^{<264} \rightarrow \{0, 1\}^{8l}$. Disini $\{0,1\}^{<264}$ menunjukkan sekumpulan pesan dimana panjangnya paling panjang adalah $2^{64} - 1$ yang sangat layak dalam praktik aplikasinya.
2. Fungsi hash ini juga merupakan fungsi hash yang luas (*wide pipe hash function*). Seperti fungsi hash lainnya, akan digunakan nilai insialisasi dan aturan *padding* yang berbeda yang memberikan fungsi hash dinamis (menghasilkan keluaran hash yang independen dari ukuran yang berbeda untuk satu pesan).

Algoritma $RCH_\ell(M)$:

- Aturan *padding* (*padding rule*)
  Akan dilakukan *pad* terhadap pesan sebagai berikut : $\text{pad}(M)=\text{bin}_8(l) \parallel M \parallel 1 \parallel 0^k \parallel \text{bin}_{64}(|M|)$, dimana $\text{bin}_{64}(|M|)$ adalah representasi bilangan biner 64 bit dari banyaknya bit M dan k adalah bilangan bulat positif paling kecil sehingga $8+|M|+1+k+64 \equiv 0 \mod 512$. $\text{Pad}(M) = M_1 \parallel \ldots \parallel M_t$ sehingga $| M_i | = 512$.
- Iterasi klasik (*iterate classical*)
  Misal $M_1 \parallel \ldots \parallel M_t$ adalah pesan yang di-*pad*. Dan $(S_0, j_0) := (S^{IV}, 0)$ merupakan nilai inisialisasi. Invoke fungsi *compression-like* C secara iteratif sama dengan iterasi klasik sebagai berikut :
  $(S_0,j_0) \xrightarrow{M_1} (S_1,j_1) \xrightarrow{M_2} \ldots (S_{t-1},j_{t-1}) \xrightarrow{M_t} (S_t,j_t) := C^+(M)$
  Mengingat bahwa, $(S, j) \rightarrow_x (S^*, j^*)$ berarti bahwa $C((S, j), X) = (S^*, j^*)$, dimana C : Perm × $[N]$ × $\{0,1\}^{512} \rightarrow$ Perm × $[N]$
- *Post-processing*
  Pengolahan tonggak / *post-processing* dibagi dalam beberapa langkah. Misalkan (S_t, j_t) adalah *internal state* setelah iterasi klasik, $C^{+'}(M) = (S_t, j_t)$.
  1. Hitung $S_{t+1} = S_0 \circ S_t$ dan $j_{t+1} = j_t$.
  2. Didefinisikan nilai hash akhir $RCH_\ell(M)$ dengan $\text{HBG}_l(\text{OWT}(S_{t+1}, j_{t+1}))$ (HBG_l dan OWT diberikan pada TABEL 1 dibawah ini).

C((S,j), X)	OWT((S,j))	HBG_ℓ((S,j))
for i = 0 to 255 j = j + S[i] + X[r(i)]; Swap(S[i],S[j]); Return (S,j);	Temp1 = S; for i = 0 to 511 j = j + S[i]; Swap(S[i],S[j]); Temp2 = S; S = Temp1 ∘ Temp2 ∘ Temp1; Return (S,j);	for i = 1 to ℓ j = j + S[i]; Swap(S[i],S[j]); Out = S[S[i] + S[j]];

4.Analisis Keamanan dan Performansi

Saya akan menjelaskan analisis keamanan terhadap preimej, preimej kedua, dan serangan *collision*. Saya juga akan menghitung banyaknya operasi dasar untuk menghitung nilai hash seperti tabel *lookup* dan penjumlahan modular.

Pilihan Nilai Insial

Telah dipilih nilai inisial S^{IV} karena tidak *b-conserving*. Kemudian akan dijelaskan apa itu *b-conserving* dan kunci *b-exact*. Sebuah kunci *b-exact* merupakan kunci terlemah dalam algoritma RC4 *key scheduling*. Telah dilakukan beberapa penelitian yang menunjukkan beberapa kelemahan kunci lemah.

Definisi :

1. jika $S[t] \equiv t \mod b$ untuk semua t, permutasi S dikatakan sebagai *b-conserving*. Jika $S(t) \equiv t \mod b$ untuk setidaknya N–2 nilai dari t, maka permutasi S mendekati *b-conserving*.
2. Misalkan b dan k adalah dua buah bilangan bulat, dan K adalah K-byte kunci. Maka K disebut kunci *b-exact* jika untuk index r manapun, $K[r \mod k] \equiv (1-r) \mod b$. Selain itu, jika $K[0] = 1$ dan $msb(K[1]) = 1$ maka K disebut kunci spesial *b-exact* dimana $msb(x)$ berarti bit paling signifikan dari x.

4.1 Daya Tahan Preimej

Diberikan nilai hash dari sebuah pesan secara acak yang dipilih dari *space* pesan, akan ditunjukkan kesulitan dalam mencari preimej. Karena dimiliki OWT, transformasi satu arah, satu-satunya yang dapat menggunakan "*meet in the middle attack*" pas setelah melakukan transformasi satu arah dan sebelum melakukan pembangkitan byte hash. Untuk lebih jelasnya, diberikan nilai hash $h = h_0 \parallel h_1 \parallel \ldots \parallel h_{t-1}$, balikkan HBG (ini mungkin karena algoritma pembangkitan byte hash mudah dibalikkan) dan simpan sekumpulan A dari pasangan (S, j) dimana menghasilkan h setelah pembangkitan byte hash. Kemudian dapat memilih pesan M secara acak dan menghitung $\text{OWT}(C^+(M))$ dan cari collision pada sekumpulan A. Tetapi kerumitan dari "*meet in the middle attack*" ini membutuhkan kira-kira $2^{1692/2} = 2^{846}$ *queries*. Dapat digunakan pendekatan berbeda dengan menggunakan *b-predictive a-state* yang akan dijelaskan sebagai berikut.

Serangan Preimej berdasarkan pada *Predictive RC4 states*

Definisi :

1. Sebuah *a-state* ditentukan sebagian RC4 *state*, termasuk *i, j*, dan elemen-elemen *a* dari S (tidak perlu berurutan). Lebih jelasnya, tuple $p = (i, j, (i_1, \ldots, i_a), (j_1, \ldots, j_a))$ merupakan sebuah keadaan *a* (*a-state*).
2. Keadaan *a* (*a-state*) $p = (i, j, (i_1, \ldots, i_a), (j_1, \ldots, j_a))$ cocok dengan RC4 *state* (i, j, S) jika $S[i_k] = j_k$ untuk $1 \le k \le a$. Dikatakan bahwa p meramalkan hasil keluaran *r th* jika untuk semua keadaan cocok dengan p, menghasilkan hasil keluaran byte yang sama setelah r putaran. Keadaan *a* (*a-state*) p dikatakan akan *b-predictive a-state* jika p meramalkan hasil keluaran $r_1 < \ldots < r_b (\le 2N)$.

Dalam sebuah serangan praktis terhadap RC4, Mantin dan Shamir menunjukkan serangan yang berbeda berdasarkan atas *b-predictive a-state* dimana membutuhkan $O(N^{2a-b+3})$ byte hasil keluaran. Kemudian, Paul dan preneel memodifikasi definisi ini dengan mempertimbangkan $1 \equiv r_1 < \ldots < r_b \leq N$. Menurut definisi ini, telah ditunjukkan bahwa *b-predictive a-state* dapat muncul hanya jika $a \geq b$. Pada analisis statistik dari pembangkitan aliran kunci RC4 dikatakan banyaknya *b-predictive b-state* spesial (dikenal sebagai keadaan kebetulan/*fortuitous state* dimana semua *b* meramalkan keadaan yang berurutan) diberikan. Catatan bahwa nilai byte keluaran hash ini berurutan.

b	Total	Total state	Prob
2	516	$2^{31,99}$	$2^{-22,9}$
3	290	$2^{39,98}$	$2^{-31,8}$
4	6540	$2^{47,97}$	$2^{-35,2}$
5	25.419	$2^{55,94}$	$2^{-41,3}$
6	101.819	$2^{63,92}$	$2^{-47,2}$

Tabel 2. Kolom yang kedua adalah banyaknya spesial *b-predictive b-state* dikenal sebagai *fortuitous state*. Total state berarti banyaknya kemungkinan pilihan yang berbeda dari *i*, *j* dan nilai-nilai dari *S* pada indeks *b* yang sesuai. Contoh, dalam kasus *b* = 2, total state adalah $256 \times 256 \times 255 \times 256 \approx 2^{31,99}$. Kemudian, $516/2^{31,99}$ adalah nilai kemungkinan bahwa keadaan acak merupakan salah satu dari *fortuitous state* dengan panjang 2.

Seandainya diberikan nilai hash yang dibangkitkan dari sebuah *b-predictive b-state* dengan beberapa pilihan *j*. berarti bahwa byte *b* hasil keluaran hash ditentukan hanya dengan elemen-elemen *b* dari *intermediate* permutasi *S* dan *j* dimana $\mathrm{OWT}(C^{+(}M))$ $=(S,j)$. Jadi hasil manapun dari $\mathrm{OWT}(C^{+(}M))$ memenuhi kondisi *b*+1 dapat menjadi sebuah preimej untuk nilai hash yang diberikan. Sekarang kemungkinan bahwa pesan acak memenuhi kondisi *b*+1 adalah $1/N^2(N\text{-}1)\ldots(N\text{-}b\text{+}1)$. sisanya memiliki *l*-*b* byte hash akan sama dengan kemungkinan $1/N^{l-b}$. Kemudian, kemungkinan untuk mendapatkan preimej menjadi $1/ N^{l-b+2}$ $(N\text{-}1)\ldots(N\text{-}b\text{+}1)$. Satu-satunya yang dapat mengecek bahwa kemungkinan kurang dari $1/N^l$ untuk $b \leq 64$. Diberikan kemungkinan untuk nilai *l* yang lebih kecil pada tabel 1. Kemudian, serangan preimej berdasarkan *frotuitous state* tidak membantu dan diperlukan N^l kerumitan.

4.2 Daya Tahan Preimej Kedua

Dalam preimej kedua pada fungis hash n-bit untuk kurang lebih dari 2^n pekerjaan, Kelsey dan Schneier menggambarkan serangan umum preimej kedua yang mengurangi kerumitan dari 2^n (kasus trivial untuk hasil keluaran *n*-bit) menuju $2^{n/2}$. Kita dapat memakai serangan mereka untuk konstruksi MD klasik yang mana mengulang fungsi kompresi sehingga panjang dari nilai intermediate sama dengan nilai hasil keluaran hash. Jika $w \geq 2n$, maka *wide pipe hash* akan aman terhadap serangan preimej kedua. Model dasar dari RCH_l mengikuti model *wide pipe hash*. Dalam kasus RCH_l, *w* bernilai sekitar 1692 bit dan hasil hash bernilai kurang dari 512 bit. Oleh karena itu, sejak kerumitan dari serangan preimej kedua adalah sekitar 2^{846} , maka dapat dikatakan bahwa RCH_l aman terhadap serangan preimej kedua.

4.3 Daya Tahan *Collision*

4.3.1 Kerumitan dari *Birthday Attack*

Misal *X* dan *Y* adalah variabel acak yang independen dan didistribusikan sama mengambil nilai dari kumpulan $R = \{r_1,\ldots,r_L\}$. dan p_i adalah kemungkinan $X = r_i$ (sama dengan $Y = r_i$). mudah untuk dicek bahwa $\Pr[X = Y] = \sum_{i=1}^{L} p_i^2$. fungsi $f : D \rightarrow R$, *x* dan *y* dipilih secara sama dan independen dari *D*. Kemudian $\Pr[f(x) = f(y)] = \sum_{i=1}^{L} p_i^2$. Demikian, kita membutuhkan sedikitnya $1/(\sum_{i=1}^{L} p_i^2)^{1/2}$ banyaknya query untuk mendapatkan sebuah collision dengan menggunakan serangan *birthday* pada fungsi *f*.

Sekarang kita pertimbangkan $D = \mathbf{Perm} \times [N]$, $R = \{0,1\}^{8l}$ dan $f : \mathbf{Perm} \times [N] \rightarrow \{0,1\}^{\delta}$ adalah fungsi HBG_{δ}. HBG_{δ} tidak lain adalah RC4-PRBG dan karena itu dipertimbangkan serangan khusus yang berbeda untuk menghitung kompleksitas serangan *birthday*.

a. Serangan khusus Mantin dan Shamir 2nd byte
   Misal $y_1,\ldots,y_{\delta}$ adalah byte keluaran PRBG. Telah ditunjukkan bahwa $j = 0$ dan *S* dipilih secara bersamaan dengan kemungkinan $y_2 = 0$ (0byte) mendekati $2/N$. Terdapat $2^{8(l-1)}$ keluaran yang memiliki nilai byte kedua 0. Asumsi bahwa seluruh keluaran sisanya mungkin sama-sama. Kita lihat bahwa kompleksitas serangan *birthday* mendekati $q=2^{4\delta} \times 2^{-.001} = 2^{4\delta-.001}$. Demikian, keamanannya kurang .001 bit dibandingkan situasi idealnya. Selain itu, diasumsikan $j = 0$. Bias untuk serangan khusus ini lebih kurang pada saat kita memiliki distribusi seragam pada *j*, seperti pada kasus kita.

b. Serangan khusus Paul dan Preneel
   Kita pelajari serangan ini dari sudut pandang kompleksitas serangan *birthday*. Pada serangan ini, telah terbukti bahwa dua byte pertama sama dengan kemungkinan mendekati $1/N(1 - 1/N)$. Satu-satunya yang dapat membuat perhitungan yang sama untuk melihat bahwa kompleksitas serangan *birthday* mendekati nilai $2^{4\delta-.00000008}$.

c. *Finney state*
   Misalkan (S_1,j_1) dan (S_2,j_2) dipilih secara bersamaan kemudian kemungkinan nilai keluaran hash akan sama $(\mathrm{HBG}_{\delta}(S_1,j_1) = \mathrm{HBG}_{\delta}(S_2,j_2))$ adalah mendekati $\frac{1}{N^4 \times N(N-1)\ldots(N-\delta+1)} + (1 - \frac{1}{N^4})\frac{1}{N^{\delta}}$.
   Ini dapat dihitung dengan mengkondisikan pada kejadian kedua state merupakan *finney state*.

Demikian, kompleksitas serangan *birthday* dapat dihitung dan kira-kira bernilai $N^{4\ell-.000001375}$.

Semua hitungan diatas adalah berdasarkan asumsi. Nilai kompleksitas yang sebenarnya mungkin berbeda tetapi tidaklah mudah untuk dihitung dimana distribusi keluaran tidak diketahui.

4.3.2 Serangan menggunakan sifat khas untuk *internal collision*

Pada seksi ini kita tulis RCH untuk menunjukkan RCH_ℓ bilamana analisis tidak bergantung pada pilihan ℓ. Serangan *collision* dipusatkan pada menemukan karakteristik dengan kemungkinan tinggi. Baru-baru ini, Wang *et al* menyarankan strategi serangan baru untuk menemukan karakteristik *collision* dengan kemungkinan tinggi dengan menggunakan tambahan(*addition*) dan perbedaan XOR. Terutama, metode serangan mereka sangat berkait dengan properti fungsi boolean yang digunakan pada setiap fungsi hash. Mereka juga menemukan *collision* dari MD4, MD5, HAVAL, SHA-0, dan menunjukkan kerumitan dari menemukan sebuah *collision* dari SHA-1 adalah 2^{63} operasi. Tidak seperti fungsi hash MD4, RCH menggunakan fungsi nonlinier, pertukaran kocok (*exchange shuffle*). Sejak proses *exchange shuffle* mencegah penambahan dan perbedaan XOR dipertahankan dan tidak ada fungsi boolean, jadi tidak dapat mengaplikasikan metode serangan Wang *et al* kedalam RCH. Oleh karena itu, kita butuh pendekatan yang berbeda untuk analisis keamanan RCH. Langkah pertama, pertimbangkan dua karakteristik dengan langkah kecil pada RCH. Misalkan x dan x' merupakan dua blok pesan dan x_i dan x_i ‘ menunjukkan byte pesan pada tingkat i.

Contoh pertama : Untuk setiap i dan $S[i]$ = a, $S[i + 1]$ = b, $x_{i+1} = x_i$, jika $i = j$ sebelum meng-*update* j dan $x_i + a \equiv 0 \bmod 256$ dan $x_{i+1} + b \equiv 0 \bmod 256$, maka final permutasi intermediate S dan j menjadi sama dengan x dan x' demikian $x'_i = x_i + 1$, $x'_{i+1} = x_{i+1}$ dan $x'_{i+2} = x_{i+2} + 255$. Disini kita membutuhkan tiga kondisi untuk membatasi nilai dari $S[i]$, $S[i+1]$ dan j.

Contoh kedua : Untuk setiap i dan $S[i] = a$, $S[i + 1]$ = b, $x_i = x_{i+1} = x_{i+2} = x_{i+3} - 4$, jika $i = j$ sebelum meng-*update* j dan $x_i + a \equiv 1 \bmod 256$ dan $a \equiv b - 1 \bmod 256$, maka final permutasi intermediate S dan j menjadi sama untuk x dan x' demikian $x'_i = x_i - 1$, $x'_{i+1} = x_{i+1}$, $x'_{i+2} = x_{i+2} - 1$, $x'_{i+3} = x_{i+3} + 2$ dan $x'_{i+4} = x_{i+4} - 3$. Disini dibutuhkan tiga kondisi untuk membatasi nilai dari $S[i]$, $S[i+1]$ dan j.

Pada contoh pertama diatas merupakan 3 langkah karakteristik dengan 3 kondisi demikian dua byte pesan berbeda untuk x dan x'. Panjang dari karakteristik sama dengan jumlah kondisi. Contoh kedua merupakan 5 langkah karakteristik dengan 3 kondisi demikian empat byte pesan berbeda untuk x dan x'. Dibutuhkan lebih banyak dan berbeda dari byte pesan untuk x dan x' dalam hal untuk mendapatkan nilai panjang dari karakteristik yang panjang dengan sedikit kondisi. Sejak RCH menggunakan tiap byte pesan empat kali dengan metode *reordering*, dalam kasus menggunakan banyak byte pesan berbeda untuk x dan x', sebuah serangan harus membuat karakteristik yang panjang dan rumit untuk rentetan yang lain. Disini, kita anggap penyerang tertentu mencoba untuk membangun karakteristik dengan begitu setiap langkah memiliki satu kondisi. Pada kasus ini, dapat dikatakan batas keamanan dari kerumitan serangan. Jika dua pesan berbeda pada posisi k_1 dan k_2 dan misal i_1 dan i_2 menjadi kebalikannya dengan mematuhi fungsi rentetan r_1 . Maka kita butuh untuk meletakkan kondisi pada $S[i_1]$, $S[i_1 + 1]$ …, $S[i_2]$ dan j. *Reordering* yang sudah dipilih memiliki properti untuk setiap k_1 dan k_2.

$$\sum_{k=1}^{3} \left| \eta_k^{-1}(k_1) - \eta_k^{-1}(k_2) \right| \geq 24$$

dan sebab itu total jumlah kondisi sedikitnya 30. Ini karena kita membutuhkan $\left| \eta_k^{-1}(k_1) - \eta_k^{-1}(k_2) \right| + 2$ kondisi untuk setiap rentetan. Kemudian, kita membutuhkan query sebanyak 2^{240} untuk menemukan *collision*. Ini merupakan uraian heuristic. Secara intuitif tidak mungkin mendapatkan *collision* dengan metode diatas dalam kerumitan ini. Faktanya, masih belum jelas bagaimana untuk membuat serangan *collision* dengan kerumitan ini.

4.3.3 Serangan menggunakan properti *b-conserving*

RCH_l memiliki inisial permutasi S^{IV} yang tidak ada properti *b-conserving*. Walaupun S^{IV} bukan *b-conserving*, permutasi intermediate dapat menjadi *b-conserving* dengan mengaplikasikan pesan yang spesifik. Jika permutasi intermediate setiap langkahnya acak, maka dapat dihitung kemungkinan adanya permutasi *b-conserving* dalam nilai intermediate untuk setiap b sebagai berikut :

a. Untuk $b = 2$, $(128!)^2 / 256! \approx 2^{-252}$
b. Untuk $b = 4$, $(64!)^2 / 256! \approx 2^{-490}$
c. Untuk $b = 8$, $(32!)^2 / 256! \approx 2^{-743}$
d. Untuk $b = 16$, $(16!)^2 / 256! \approx 2^{-976}$

Dalam hal untuk mendapatkan 2-*conserving* permutasi intermediate, dibutuhkan 2^{252} query C kemudian kita dapat memilih blok-blok pesan demikian seluruh permutasi intermediate yang maju adalah hampir 2-*conserving* dengan kemungkinan 2/5. Oleh karena itu, kita dapat mengurangi ukuran nilai intermediate dari 1684 bit (dapat disamakan dengan 256!) menjadi setidaknya 1432 bit (terdapat 128!×128! permutasi 2-*conserving*) sehingga kita dapat menemukan sebuah *collision* pada nilai intermediate dengan kompleksitas sebesar 2^{176} dimana lebih banyak dari nlai tersebut untuk serangan *collision* dengan nilai keluaran hash kurang

dari 512 bit. Dalam kasus lain memiliki kemumgkinan yang sangat kecil sehingga dapat kita abaikan.

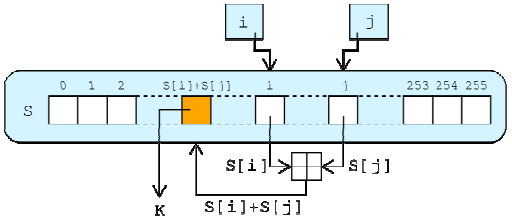
5.KESIMPULAN DAN SARAN

Pada makalah ini,saya merepresentasikan fungsi hash baru RC4-Hash, dan saya berpendapat bahwa fungsi hash ini aman dan cepat. Fungsi hash ini didasarkan pada struktur sederhana dari RC4 dimana algoritma RC4 tersebut sangatlah cepat. Untuk setiap 512 bit pesan, dibutuhkan 1024 modulo pertambahan dan 1536 lookup(untuk menghitung $C^{+}(.)$). Proses *post processing* sedikit lebih mahal tetapi tidak akan menjadi masalah jika memiliki pesan hash yang panjang yang diaplikasikan pada setiap pesan. Dalam *post processing* dimiliki tambahan 512+l dan lookup 2048+l. Fungsi hash ini membangkitkan keluaran nilai hash yang berubah-ubah. Struktur fungsi tersebut berbeda dari fungsi hash lainnya yang diketahui. Dilihat dari struktur internal baru dan ukuran dari state internal yang besar (kira-kira 1700 bit) dapat melawan serangan-serangan umum seperti serangan *path breaking*. Fungi hash ini sangat mudah diimplementasi dan efisien dalam perangkat lunak dan juga cocok dengan berbagai tingkat keamanan. Saya harap fungsi hash ini akan sangat berguna kedepannya. Catatan bahwa RC4 berdasar pada 8 bit aritmatik, tetapi ada juga yang memanfaatkannya pada 32/64 bit arsitektur mesin zaman sekarang dengan kecepatan yang dipertinggi. Fungsi hash baru ini belumlah sempurna. Performansi fungsi hash baru ini masih dibawah 1,5 kali kecepatan SHA-1. Mungkin saja ada yang ingin mendesain fungsi hash baru berdasarkan RC4 dengan keamanan yang lebih kuat dan dengan menambahkan kecepatannya.

DAFTAR REFERENSI

http://en.wikipedia.org/wiki/RC4
http://en.wikipedia.org/wiki/Cryptographic_hash_function
http://en.wikipedia.org/wiki/Collision_attack
http://en.wikipedia.org/wiki/Meet-in-the-middle_attack
http://en.wikipedia.org/wiki/Hash_collision
http://en.wikipedia.org/wiki/Preimage_attack
http://www.burtleburtle.net/bob/hash/#lookup
http://209.85.175.104/search?q=cache:sc1FaFGKOxoJ:delta.cs.cinvestav.mx/~nandi/lecture_18_06/RC4_indocrypt06_ver1.pdf+rc4,hash&hl=id&ct=clnk&cd=1&gl=id

LAMPIRAN



Diatas adalah gambar pembangkitan kunci aliran *K*.

Saya gambarkan *reordering* yang digunakan pada algoritma hash. Digunakan fungsi identiti untuk r_0 dan r_i yang didefinisikan nanti untuk $1 \le i \le 3$. Fungsi r yang terbatas pada $[64_i, 64_i + 63]$ tidak lain adalah r_i ,$0 \le i \le 3$

r_1:0,55,46,37,28,19,10,1,56,47,38,29,20,11,2,57,48,39,30,21,12,3,58,49,40,31,22,13,4,59,50,41,32,23,14,5,60,51,42,33,24,15,6,61,52,43,34,25,16,7,62,53,44,35,26,17,8,63,54,45,36,27,18,9.

r_2:0,57,50,43,36,29,22,15,8,1,58,51,44,37,30,23,16,9,2,59,52,45,38,31,24,17,10,3,60,53,46,39,32,25,18,11,4,61,54,47,40,33,26,19,12,5,62,55,48,41,34,27,20,13,6,63,56,49,42,35,28,21,14,7.

r_3:0,47,30,13,60,43,26,9,56,39,22,5,52,35,18,1,48,31,14,61,44,27,10,57,40,23,6,53,36,19,2,49,32,15,62,45,28,11,58,41,24,7,54,37,20,3,50,33,16,63,46,29,12,59,42,25,8,55,38,21,4,51,34,17.

Initial value permutasi atau S^{IV} :

145, 57, 133, 33, 65, 49, 83, 61, 113, 171, 63, 155, 74, 50, 132, 248, 236, 218, 192, 217, 23, 36, 79, 72, 53, 210, 38, 59, 54, 208, 185, 12, 233, 189, 159, 169, 240, 156, 184, 200, 209, 173, 20, 252, 96, 211, 143, 101, 44, 223, 118, 1, 232, 35, 239, 9, 114, 109, 161, 183, 88, 66, 219, 78, 157, 174, 187, 193, 199, 99, 52, 120, 89, 166, 18, 76, 241, 13, 225, 6, 146, 151, 207, 177, 103, 45, 148, 32, 29, 234, 7, 16, 19, 91, 108, 186, 116, 62, 203, 158, 180, 149, 67, 105, 247, 3, 128, 215, 121, 127, 179, 175, 251, 104, 246, 98, 140, 11, 134, 221, 24, 69, 190, 154, 253, 168, 68, 230, 58, 153, 188, 224, 100, 129, 124, 162, 15, 117, 231, 150, 237, 64, 22, 152, 165, 235, 227, 139, 201, 84, 213, 77, 80, 197, 250, 126, 202, 39, 0, 94, 42, 243, 228, 87, 82, 27, 141, 60, 160, 46, 125, 112, 181, 242, 167, 92, 198, 172, 170, 55, 115, 30, 107, 17, 56, 31, 135, 229, 40, 111, 37, 222, 182, 25, 43, 119, 244, 191, 122, 102, 21, 93, 97, 131, 164, 10, 130, 47, 176, 238, 212, 144, 41, 14, 249, 220, 34, 136, 71, 48, 142, 73, 123, 204, 206, 4, 216, 196, 214, 137, 255, 195, 26, 8, 51, 178, 2, 138, 254, 90, 194, 81, 245, 106, 95, 75, 86, 163, 205, 70, 226, 28, 147, 85, 5, 110.