Study Mengenai Algoritma GOST Hash

Mochamad Reza Akbar – NIM: 13507131
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
If17131@students.if.itb.ac.id

***Abstract*—Kriptografi adalah salah satu pendekatan yang efektif untuk mendukung keamanan data. Penggunaannya sudah banyak diaplikasikan diberbagai macam bidang termasuk dalam wilayah pemerintahan maupun aktivitas komersial.**

Fungsi hash pada kriptografi merupakan sebuah prosedur yang mengambil blok-blok data sebuah pesan dan mengembalikannya menjadi sebuah pesan String bit dengan panjang yang telah ditentukan (fixed-size). Fungsi hash banyak digunakan untuk berbagai jenis aplikasi untuk keamanan informasi seperti digital signature, message authentication codes (MACs). Fungsi ini juga dapat digunakan untuk sebagai checksum untuk mendeteksi corrupt data dan sebagainya.

Fungsi GOST hash yang merupakan dasar pembuatan GOST block cipher dikembangkan oleh Federal Agency for Government Communication and Information dan oleh All-Russia Scientific and Research Institute of Standardization yang kemudian dipakai sebagai fungsi hash standar di Rusia. Fungsi ini dikembangkan untuk memperluas aplikasi teknologi informasi dalam hal membuat, memproses dan menyimpan dokumen yang didalamnya dibutuhkan konsistensi conten, maintenance dan autentisi.

***Kata Kunci* — fixed-size, GOST hash, fungsi hash, kriptografi.**

I. PENDAHULUAN

Fungsi hash pada kriptografi (H) memetakan pesan M yang memiliki berbagai macam panjang pesan menjadi sebuah nilai hash yang memiliki panjang tetap. Secara formal, fungsi hash yang digunakan harus bisa memenuhi beberapa kebutuhan keamanan:

- Collision resistance: dua pesan yang berbeda harus memiliki nilai hash yang berbeda. Sebuah pesan (M1) yang dipetakan dengan fungsi hash (H) memiliki nilai hash yang tidak mungkin dimiliki oleh pesan lain (M2) yang dipetakan dengan fungsi hash yang sama (H).
- Second preimage resistance: jika terdapat sebuah pesan M, maka tidak mungkin menemukan sebuah pesan yang berbeda jika terdapat H(M1) = H(M2).
- Preimage resistance: nilai hash yang dihasilkan dari fungsi hash, tidak dapat dikembalikan menjadi pesan semula. Pesan yang telah di-hash tidak dapat dikembalikan seperti semula.

Semua sifat ini secara tidak langsung menyatakan bahwa segala macam lawan yang jahat atau serangan tidak dapat mengganti atau merubah pesan tanpa merubah nilai hashnya.

Serangan yang dilakukan kriptanalis terhadap fungsi hash lebih focus terhadap terhadap serangan collision attack. Tabrakan atau perbedaan biasanya terjadi pada penggunaan fungsi hash, akan tetapi kita tidak akan menyadari terjadinya collision attack pada fungsi GOST hash. Pada makalah ini, saya akan menjelaskan mengenai fungsi GOST hash. Fungsi GOST hash adalah fungsi yang digunakan banyak orang di Rusia dan secara spesifik Rusia membuat sebuah fungsi hash standar Russian national standard GOST 34.11-94. Fungsi standar ini dikembangkan oleh GUBS of Federal Agency Goverbnebt Communication and Information dan All-Russian Scientific and Research Institute of Standardization. Fungsi GOST hash merupakan satu-satunya fungsi hash yang dapat digunakan untuk Russian digital signature algorithm GOST 34.10-94. Selain itu fungsi GOST hash ini juga digunakan di beberapa aplikasi lainnya.

Fungsi GOST hash memproses pesan dengan berbagai macam panjang menjadi sebuah keluaran dengan panjang yang telah ditetapkan sepanjang 256 bit. Pesan masukan akan ditambahkan dengan bilangan nol sampai panjang pesan merupakan kelipatan 256 bit. Bit-bit terakhir akan diisi dengan panjan pesan yang akan di-hash.

Berikut adalah contoh fungsi GOST hash pesan kosong dan pesan biasa dengan perubahan kecil terhadap pesan:

```
GOST("") =
ce85b99cc46752fffee35cab9a7b0278ab
b4c2d2055cff685af4912c49490f8d


GOST("The quick brown fox jumps
over the lazy dog") =
77b7fa410c9ac58a25f49bca7d0468c929
6529315eaca76bd1a10f376d1f4294

GOST("The quick brown fox jumps
over the lazy cog") =
a3ebc4daaab78b0be131dab5737a7f67e6
02670d543521319150d2e14eeec445
```

Berikut adalah algoritma implementasi fungsi GOST

Hash (http://www.autochthonous.org/crypto/gosthash.tar.gz.):

Gosthash.h

```
#ifndef GOSTHASH_H
#define GOSTHASH_H

#include <stdlib.h>

/* State structure */

typedef struct
{
  unsigned long sum[8];
  unsigned long hash[8];
  unsigned long len[8];
  unsigned char partial[32];
  size_t partial_bytes;
} GostHashCtx;

/* Compute some lookup-tables that are needed by
all other functions. */

void gosthash_init();

/* Clear the state of the given context structure. */

void gosthash_reset(GostHashCtx *ctx);

/* Mix in len bytes of data for the given buffer. */

void gosthash_update(GostHashCtx *ctx, const
unsigned char *buf, size_t len);

/* Compute and save the 32-byte digest. */

void gosthash_final(GostHashCtx *ctx, unsigned
char *digest);

#endif /* GOSTHASH_H */
```

II. FUNGSI GOST HASH

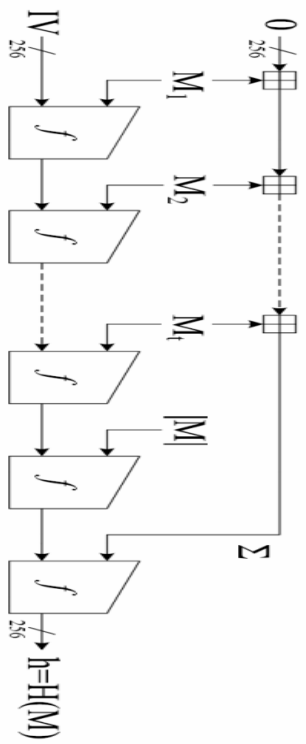
Pesan dengan blok-blok yang berukuran 256 akan diproses oleh fungsi GOST hash menjadi nilai hash 256 bit. Jika panjang pesan tidak mencapai keliapatan 256, pesan akan di-padding seminimal mungkin hingga kondisi tercapai (panjang pesan sama dengan kelipatan 256 bit).

$$H_0 = IV \quad (1)$$

$$H_i = f\,(H_{i-1}, M_i) \text{ for } 0 < i \leq t \quad (2)$$

$$H_{t+1} = f\,(H_t, |M|) \quad (3)$$

$$H_{t+2} = f\,(h_{t+1}, \Sigma) = h, \quad (4)$$



Gambar 1. Struktur fungsi GOST hash

Setelah di-padding, pesan M akan dibagi menjadi t-blok pesan ($M = M_1||M_2||M_3||..||M_t$). Nilai hash yang dihasilkan akan diproses seperti pada Gambar 1.

Dimana $\Sigma = M_1 \square M_2 \square .. \square M_t$, dan $\square$ merupakan hasil modulo 2^{256} setelah penjumlahan. IV adalah initial value yang telah didefiniskan sebelum proses dilakukan dan |M| merepresentasikan sebagai bit panjang yang ditambahkan diakhir pesan. Seperti yang terlihat pada persamaan (4), checksum (Σ) merupakan hasil modulo dari penjumlahan semual blok-blok pesan. fungsi GOST hash. Penggunaan komputasi checksum ini merupakan bagian penting yang digunakan pada fungsi GOST hash dibandingkan MD5 dan SHA-1.

Fungsi kompresi pada fungsi f, pada dasarnya merupakan gabungan dari tiga bagian: state update transformation, the key generation dan the output transformation, seperti yang terlihat pada gambar 2.

2.1 State Update Transformation

The state update transformation pada GOST terdiri dari empat hal GOST blok cipher secara parallel, yang ditunjukkan oleh E. Nilai hash pada pertenganan H_{i-1} akan dibagi menjadi empat $h_3||h_2||h_1||h_0$ sebanyak 64 bit kata. Setiap 64 bit kata akan digunakan sebagai satu aliran state update transformation untuk konstruksi 256 bit $S = s_3||s_2||s_1||s_0$ sebagai berikut:

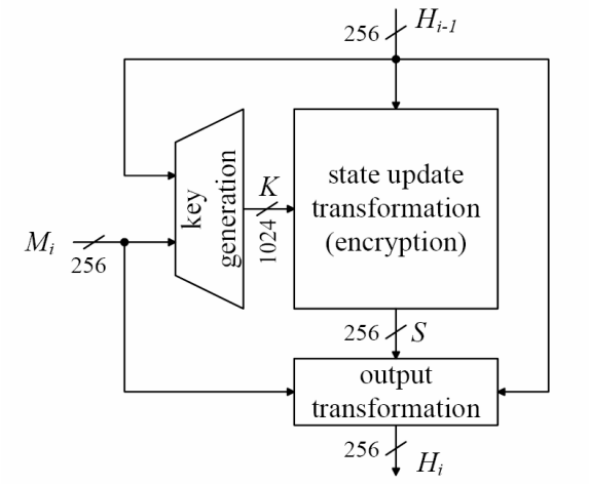
$$s_0 = E\,(k_0, h_0) \quad (5)$$

$$s_1 = E\,(k_1, h_1) \quad (6)$$

$$s_2 = E\,(k_2, h_2) \quad (7)$$

$s_3 = E (k_3,h_3)$ (8)

Dimana E(K,P) menunjukkan ekripsi dari 64 bit plaintek P dengan menggunakan 256 bit kunci K.



Gambar 2. The compression function of GOST

2.2 Key Generation

Key generation pada GOST digunakan sebagai masukan tengah-tengah nilai hash H_{i-1} dan blok pesan M_i untuk mengkomputasi 1024 bit kunci K. kunci ini akan dibagi menjadi empat kunci 256 bit. $K = k_3||k_2||k_1||k_0$, dimana tiap kunci k_i digunakan sebagai satu aliran sebagai kunci untuk GOST block cipher E pada state update transformation. Keempat kunci didapat dari hasil komputasi sebagai berikut:

$k_0 = P (H_{i-1} \square M_i)$ (9)

$k_1 = P (A (H_{i-1}) \square A_2 (M_i))$ (10)

$k_2 = P (A_2 (H_{i-1}) \square Const \square A_4 (M_i))$ (11)

$k_3 = P (A (A_2 (H_{i-1}) \square Const) \square A_6 (M_i))$ (12)

dimana A dan P adalah linier transformations dan Const adalah sebuah konstanta. Perlu diperhatikan bahwa $A_2 (x) = A (A (x))$. Sebagai definisi linier transformation A dan P seperti nilai dari konstanta Const.

2.3 Output Transformation

Output transformation dari GOST merupakan gabungan dari tengah-tengah nilai hash H_{i-1}, blok pesan M_i dan keluaran dari state update transformation S untuk mengkomputasi keluaran dari nilai H_i dari fungsi kompresi. Dan di definisikan sebagai berikut:

$H_i = \psi^{61} (H_{i-1} \square \psi (M_i \square \psi^{12} (S)))$ (13)

Linier transformation $\psi : \{0,1\}^{256}$ didapat dari:

$\Psi(\Gamma)=(\gamma_0 \square \gamma_1 \square \gamma_2 \square \gamma_3 \square \gamma_{12} \square \gamma_{15})||\gamma_{15}||\gamma_{14}||\cdot\cdot||\gamma_1$ (14)

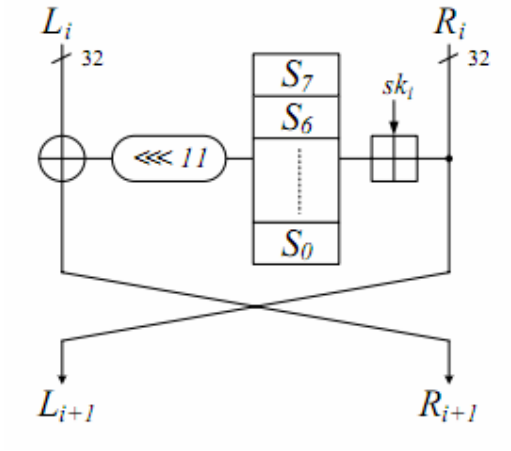
Dimana Γ dibagi menjadi 16 bit kata-kata $\Gamma = \gamma_{15}|| \gamma_{14}|| \gamma_{13}|| \gamma_{12}||..|| \gamma_0$.

III. GOST BLOCK CIPHER

Blok cipher GOST dispesifikan sebagai standar GOST 28147-89 oleh oleh pemerintahan Rusia. Berbagai macam percobaan telah dilakukan oleh berbagai kriptanalis terhadat blok cipher GOST. Akn tetapi, jika blok cipher digunakan pada fungsi hash maka skenario serangan yang diahadapi akan berbeda: penyerang memiliki kontrol penuh terhadap kunci. Hasil pertama mempertimbangkan fakta tersebut untuk analisi keamanan pada fungsi hash.

3.1 Deskripsi blok cipher

GOST blok cipher adalah 32 putaran jaringan Feistel dengan ukuran blok 64 bit dan panjang kunci 256 bit. Fungsi putaran GOST blok cipher terdiri dari punjumlahan kunci, delapan S-box berukuran 4x4 yang berbeda-beda dan rotasi perputaran, seperti yang terlihat pada Gambar 3.



Gambar 3. One Round of the GOST block cipher

Penjadwalan kunci pada GOST blok cipher didefinisikan sebagai subkunci sk_i yang diperoleh dari 256 bit $K = k_7||k_6||k_5||\ldots||k_0$ sebagai berikut:

$$ski = \begin{cases} ki \bmod 8, & i = 0,\ldots,23, \\ k7 - (i \bmod 8), & i = 24,\ldots, \end{cases} \quad (15)$$

3.2 Constructing a Fixed-Point

Observation 1 *asumsikan kita diberi plaintek P = $L_0||R_0$ dengan $L_0 = R_0$. Maka kita dapat mengkonstruksikan fixed-point untuk blok cipher dengan mengkonstuksikan fixed-point pada pertama kali setelah 8 putaran.*

Kita mengarahkan kepada plaintek $P = L_0||R_0$ dengan $L_0=R_0$ sebagai palintek yang simetri. Perhatikan bahwa dengan menggunakan blok cipher untuk mengkonstruksikan sebuah fungsi hash, penyerang memiliki kontrol penuh terhadap kunci. Selanjutnya, setiap kata pada kunci adalah satu-satunya yang digunakan pada pertama kali setelah 8 putaran blok kunci.

Karena itu, konstuksi sebuah fixed-point pada pertama setelah 8 putaran dapat deiselesaikan dengan efisien. Pertama-tama, kita memilih nilai random untuk pertama setelah 6 kata pada kunci (subkunci $sk_0,\ldots,sk_5$) dan komputasi L_6 dan R_6. Kemudian, kita memilih 2 kata terakhir pada kunci (sk_6 dan sk_7) sehingga $L_8 = L_0$ dan $R_8 = R_0$. Dengan metode ini, kita dapat mengkonstuksi sebuah fixed-point pada 8 putaran pertama pada blok cipher dengan harga komputasional sebanyak 8 putaran.

IV. Collision Attack Pada GOST

Pada bagian ini, kita akan melihat sebuah collision attack pada fungsi kompresi GOST. Collision ini terjadi dengan melihat kelemahan yang ada pada structural fungsi kompresi yang digunakan. Collision attack pada fungsi GOST hash terdapat 2^{105} evaluasi fungsi kompresi.

4.1 Konstruksi Collision Fungsi Kompresi

Mari kita lihat collision attack fungsi kompresi GOST:

$$H_i = \psi^{61}(H_{i-1}) \square \psi^{62}(M_i) \square \psi^{74}(S) \quad (16)$$

$$\underbrace{\psi^{-74}(H_i)}_{X} = \underbrace{\psi^{-13}(H_{i-1})}_{Y} \oplus \underbrace{\psi^{-12}(M_i)}_{Z} \oplus S \quad (17)$$

Karena transformasi ψ adalah liniar maka persamaan (13) dapat ditulis sebagai persamaan (16). Selanjutnya, karena ψ invertible, maka persamaan (16) dapat ditulis menajdi persamaan (17). Perlu diperhatikan bahwa Y akan berhantung terhadap H_{i-1} dan Z bergantung pada M_i, sedangkan S bergantung pada keduanya yang diproses oleh block cipher E. Selanjutnya, 256 bit X,Y dan Z akan dibagi menjadi empat bagian sebesar 64 bit kata:

$X = x_3 \| x_2 \| x_1 \| x_0$

$Y = y_3 \| y_2 \| y_1 \| x_0$

$Z = z_3 \| z_2 \| z_1 \| x_0$

Sehingga persamaan (17) akan menjadi 4 persamaan:

$$x_0 = y_0 \square z_0 \square s_0 \quad (18)$$

$$x_1 = y_1 \square z_1 \square s_1 \quad (19)$$

$$x_2 = y_2 \square z_2 \square s_2 \quad (20)$$

$$x_3 = y_3 \square z_3 \square s_3 \quad (21)$$

Selanjutnya kita asumsikan bahwa kita dapat menemukan 2^{96} blok pesan , dimana dengan $k \neq t$, semua blok-blok pesan akan menghasilkan nilai x_0 yang sama. Maka kita tahu menurut birthday paradox, 2 dari blok-blok pesan juga akan menghasilkan nilai yang sama x_1, x_2, x_3. Maka dari itu, kita harus membuat fungsi kompresi GOST yang memiliki komplektisitas evaluasi fungsi kompresi sebesar 2^{96}.

Berdasarkan penjelasan singkat tadi, kita akan mengkonstruksi blok pesan , untuk mendapatkan nilai x_0 yang sama. Asumsikan kita menginginkan nilai s_0 pada persamaan (18) menjadi sebuah constant. Karena $s_0 = E(k_0,h_0)$ dan k_0 bergantung secara linier terhadap blok pesan M_i, kita akan menemukan kunci dan blok pesan , yang semuanya menghasilkan nilai s_0 sama. Hal ini dapat diselesaikan dengan mengeksploitasi fakta bahwa pada fixed-point blok cipher GOST dapat dikonstruksi secara efisien untuk plaintek simetris. Dengan kata lain, jika h_0 adalah simetris maka kita dapat konstruksi 2^{96} blok pesan dimana $s_0 = h_0$ dan persamaan (18) menjadi

$$x_0 = y_0 \square z_0 \square h_0 \quad (22)$$

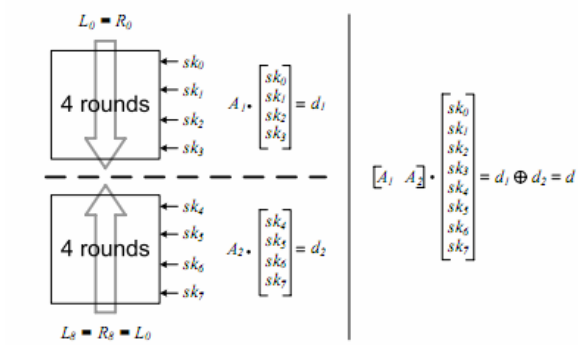
Akan tetapi, untuk menemukan blok-blok pesan untuk x0 harus memiliki nilai yang sama, kita harus tetap memastikan bahwa $y_0 \square z_0$ pada persamaan (22) memiliki nilai yang sama untuk semua blok-blok pesan. Untuk itu, kita mendapatkan persamaan (64 persamaan pada GF(2))

$$y_0 \square z_0 = c \quad (23)$$

dimana c adalah sebuah nilai asal 64 bit. Kita tahu bahwa y_0 bergantung secara linier pada H_{i-1} dan z_0 bergantung secara linier pada M_i. Untuk itu, pemilihan blok pesan M_i dan yang bersesuaian, pemilihan kunci k_0, dibatasi dengan 64 persamaan pada GF(2). Karena itu, untuk konstruksi sebuah fixed-point pada blok cipher GOST, kita harus mengetahui larangan-larangannya. Untuk selanjutnya kita biarkan

$$A.k_0 = d \quad (24)$$

Kumpulan 64 persamaan pada GF(2) yang membatasi pilihan kunci k0 menunjukkan, dimana A adalah 64 x 256 matrik pada GF(2) dan d adalah 64 bit vector. Ini mengikuti observasi sebelumnya yang pada konstruksi sebuah fixed-point pada 8 putaran pertama. Karena itu, satu metode untuk konstruksi sebuah fixed-point yang tepat adalah melakukan kostruksi banyak fixed-point yang adal dan melakukan check jika persamaan (24) cocok. Karena kita membutuhkan 2^{96} fixed-point untuk collision attack, dengan metode ini kita akan mendapatkan 2^{160} evaluasi fungsi kompresi GOST. Meskipun begitu, kita dapat meningkatkan kompleksitas dengan menggunakan meet-in-middle approach, dapat dilihat pada Gambar 4.



Gambar 4. Konstruksi fixed-point pada blok cipher GOST

Delapan putaran pertama blok cipher GOST dibagi menjadi 2 bagian P_1 (putaran 1-4) dan P_2 (putaran 5-8). Karena subkunci pada 8 putaran pertama dibatasi denhan $A.k_0$, kita juga akan membagi system 64 persamaan pada GF(2) menjadi 2 bagian:

$$A_1 \cdot \begin{bmatrix} sk_0 \\ sk_1 \\ sk_2 \\ sk_3 \end{bmatrix} = d_1 \qquad A_2 \cdot \begin{bmatrix} sk_4 \\ sk_5 \\ sk_6 \\ sk_7 \end{bmatrix} = d_2 \tag{25}$$

Dimana A = [A1,A2] dan d = d_1 □ d_2. Sekarang kita dapat mengaplikasikan serangan meet-in-mniddle untuk konstruksi 2^{64} fixed-point yang tepat untuk blok cipher GOST dengan kompleksitas 2^{64}. Ini dapat dirangkum sebagai berikut:

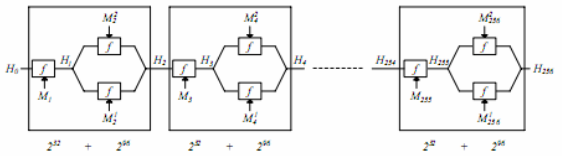
1. Pilih nilai d1 secara random. Ini juga menentukan d2=d □d1.
2. Untuk semua 2^{64} subkunci $sk_0,\ldots,sk_3$ yang memenuhi persamaan (25) komputasi L_4,R_4 dan simpan hasilnya pada list L.
3. Untuk semua 2^{64} subkunci $sk_4,\ldots,sk_7$ yang memenuhi persamaan (25) komputasi putaran 4-8 secara terbalik untuk mendapatkan L_4,R_4 dan cek untuk mencocokkan dengan entry di list L. perhatikan bahwa karena terdapat 2^{64} entri pada list L, diharapkan untuk selalu menemukan kecocokan pada entry di list L. karena itu, kita mendapatkan 2^{64} fixed-point yang cocok untuk blok cipher GOST dengan kompleksitas sekitar 2^{64} dan kebutuhan memory $2^{64}.40 \approx 2^{70}$ byte.

Dengan mengulangi serangan ini sebanyak 2^{32} kali untuk pilihan d1 yang berbeda-beda, kita mendapatkan 2^{96} fixed-point yang cocok. Dengan kata lain, kita menemukan 2^{96} kunci k_0 yang semuanya menghasilkan nilai $s_0 = E(\ ,h_0)$ sama dan juga memenuhi persamaan (24). Akibatnya, kita mendapatkan 2_{96} blok pesan yang semua hasilnya x_0 sama dengan $X = \psi^{-74}(H_i)$ dengan $k \neq t$ dimana x_1,x_2,x_3 sama. Dengan kata lain, kita dapat menemukan collision pada fungsi kompresi GOST dengan kompleksitas sekitar 2^{96} bahkan 2^{128} evaluasi fungsi kompresi GOST.

4.2 Konstruksi Collision Fungsi Hash

Collision pada fungsi kompresi dapat meluas hingga ke fungsi hash.

Multicollision adalah satuan pesan yang memiliki panjang sama yang semuanya menuju nilai hash yang sama. Seperti yang dijelaskan pada sub (4.1) fungsi kompresi GOST dapat dikonstruksi dengan kompleksitas 2^{96} jika h_0 simetris di $H_{i-1} = h_3||h_2||h_1||h_0$. Perhatikan bahwa dengan menggunakan tambahan blok pesan Mi-1 kita menemukan rantai variable $H_{i-1} = f(H_{i-2},M_{i-1})$, dimana h0 adalah simetris dengan kompleksitas 2^{32} evaluasi fungsi kompresi. Karena itu, kita dapat konstruksi 2128 collision dengan kompleksitas sekitar $2^{128}(2^{96}+2^{32}) \approx 2^{103}$ evaluasi fungsi kompresi GOST. Dengan metode ini kita mendapatkan 2^{128} pesan M* yang semuanya menuju pada nilai yang sama H_{256} seperti digambarkan pada Gambar 5.



Gambar 5. Multicollision for GOST

Untuk konstruksi collision pada checksum, kita harus menemukan 2 pesan yang jelas yang menghasilkan nilai yang sama $\Sigma = M_1 \square M_2^{j2} \square \cdots \square M_{255} \square M_{256}^{j256}$ dengan $j_2,j_4,..j_{256} \square P\{1,2\}$. Dengan memakai birthday attack, kita dapat menemukan 2 pesan dengan kompleksitas sekitaw 2^{127} tambahan pada GD(256) dan kebutuhan memory sekitar 2^{134} byte. Melihat kompleksitas yang tinggi dan kebutuhan memory birthday attack, bagian ini juga dapat menjadi serangan bootleneck. Bagaimanapun, runtime dan kebutuhan memory secara signifikan dapat berkurang dengan menggunakan menyamaratakan birthday attack yang dikenalkan Wagner di referensi [6]. Wagner memperlihatkan bahwa jika l adalah kekuatan keduanya maka kebutuhan memory dan runtime untuk menyamaratakan birthday problem diberikan dengan $2^{n/(1+lgl)}$ dan $l.2^{n/(l+lgl)}$, berturut-turut. Perhatikan bahwa standar birthday attack adalah $l = 2^1$.

Dengan mempertimbangkan kasus $l=2^3$. Maka birthday attack pada serangan bagian kedua memiliki kompleksitas $2^3.2^{235/4} = 2^{67}$ dan menggunakan lists of sizes 256/4 $=2^{64}$. Secara detail, kita harus mengkonstruksi 8 lists of size 2^{64} pada serangan langkah pertama.karena itu, kita harus mengkonstruksi $2^{8.64}$ collision pada serangan bagian pertama untuk mendapatkan 8 lists ukuranyang dibutuhkan. Mengkonstruksikan multicollision ini memiliki kompleksitas sekitar $8.64.(2^{32}+2^{96})=2105$ valuasi fungsi kkompresi dan kebutuhan memory $8.64.(2.64)=2^{16}$ byte. Karena itu, kita dapat mengkonstruksi collision untuk fungsi hash GOST dengan kompleksitas sekitar 2^{105} dan kebutuhan memory $2^{64}.2^6 = 2^{70}$ bye dengan menggunakan menyamaratakan birthday attack dengan l=8 list. Selanjutnya, colliding pasangan pesan terdiri $8.(2.64)=1024$ blok pesan. Perhatikan bahwa l=8 adalah pilihan yang terbaik untuk serangan. Pada suatu saat jika kita memilih l>8 maka kebutuhan memory serangan akan berkurang tapi

kompleksitas serangan akan meningkat. Sejak kita membutuhkan sekitar 2^{70} byte memory untuk konstruksi fixed-point pada blok cipher GOST, ini tidak memperbaiki serangan. Pada suatu saat jika kita memilih l<8 maka kebutuhan memory serangan akan meningkat secara signifikan.

Remark pada panjangnya perpanjangan property. Sekali kita menemukan collision, kita dapat mengkonstruksi banyak collision dengan menambahkan blok pesan asal. Perhatikan bahwa ini tidak memerlukan keadaan untuk straight-forward birthday attack. Dengan menggunakan birthday attack kita konstruksi collision pada nilai hash terakhir dan menambahkan blok-blok pesan akan mustahil. Karena itu, kita membutuhkan collision pada bagian iterative sebaik pada checksum untuk perpanjangan property. Perhatikan bahwa menggabungkan birthday attack umum dan multicollision, salah satunya bisa menggkonstruksi collision pada kedua bagian dengan kompleksitas sekita $128.2^{128} = 2^{135}$ ketika serangan kita memiliki kompleksitas 2^{105}.

Remark pada meaningful collisions. Pada pengaturan pemilihan awalan, penyerang mencari pasangan pesan (M,M*) sehingga jika diberikan fungsi hash H

$$H(M_{pre}||M)=H(M^*_{pre}||M^*) \quad (26)$$

Pada keadaan GOST, serangan collision pada pengaturan pemilihan awalan memiliki kompleksitas yang sama seperti collision attack. Melihat sifat generic collision attack, perbedaan pada rantai variabel-variabel dapat dibatalkan secara efisien. Asumsikan bahwa awalan yang dipilih (M_{pre},M^*_{pre}) memiliki blok pesan yang dihasilkan pada serangkaian variabel-variabel H_t dan H^*_t. maka serangan dapat dirangkum sebagai berikut:

1. Kita harus menemukan dua blok pesan $M_{t+1} = h_3||h_2||h_1||h_0$ dan $H^*_{t+1}=h^*_3||h^*_2||h^*_1||h^*_0$. Ini memiliki kompleksitas sekitas 2.2^{64} evaluasi fungsi kompresi GOST.
2. Selanjutnya kita harus menemukan dua blok pesan M_{t+2} dan M^*_{t+2} sehingga $H_{t+2}=H^*_{t+2}$. Ini dapat diselesaikan mirip dengan konstruksi collision pada fungsi kompresi GOST. Pertama, kita memilih nilai random untuk c pada persamaan (22) dan mengkonstruksi 2_{96} blok-blok pesan M^*_{t+2}, dimana $x^*_0 = x_0$. Untuk memastikan $x^*_0 = x_0$ kita harus mengatur c* pada persamaan (22) sehingga persamaan menjadi:

   $$x_0=x^*_0=y^*_0 \boxplus z^*_0 \boxplus h^*_0=c^* \boxplus h^*_0$$

   Dengan menggunakan meet-in-the-middle attack kita dapat menemukan dua blok-blok pesan M_{t+2} dan M^*_{t+2} yang menghasilkan serangkaian variabel yang sama ($H_{t+2}=H^*_{t+2}$). Langkah serangan ini memiliki kompleksitas 2.2^{96} evaluasi fungsi kompresi GOST.
3. Sekali kita telah mengkonstruksi collision pada bagian iterative. Kita harus mengkonstruksi collision pada checksum juga. Untuk itu, kita melakukan proses seperti dijelaskan sebelumnya dengan membangkitkan 2^{512} collsision dan menggunakan birthday attack dengan l=8. Kita dapat mengkonstruksi collision pada checksum GOST dengan kompleksitas 2^{105} evaluasi fungsi kompresi dan kebutuhan memory 2^{70} byte.

Karena itu kita dapat mengkonstruksi collision yang memiliki arti. Untuk fungsi hash GOST dengan kompleksitas sekitar 2^{105} evaluasi fungsi kompresi.

V. CONCLUSION

Pada fungsi GOST hash dapat dilakukan collision attack. Collision attack dilakukan berdasarkan fungsi GOST hash menggunakan blok cipher GOST yang lebih mudah diserang. Terdapat struktur pada blok cipher yang dapat dikonstruksi. Struktur internal fungsi kompresi dapat melakukan konstruksi dengan kompleksitas 2^{96} evaluasi fungsi kompresi. Meskipun kompleksitas attack yang dilakukan sangat jauh daripada aplikasi secara langsung. Makalah ini dapat memperlihatkan kelemahan yang ada pada fungsi GOST hash

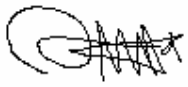
REFERENCES

[1] Munir, Rinaldi. (2004). Bahan Kuliah IF5054 Kriptografi, Departemen Teknik Informatika, Institut Teknologi Bandung.
[2] Cryptanaliysis of the GOST Hash Function http://wiki.uni.lu/esc/docs/mendel_gost.pdf.
[3] Source Code http://www.autochthonous.org/crypto/gosthash.tar.gz.
[4] http://www.rfc-archive.org/getrfc.php?rfc=5831.
[5] http://tools.ietf.org/html/draft-dolmatov-cryptocom-gost341194-07.
[6] DavidWagner.AGeneralizedBirthdayProblem.InMotiYung,editor, CRYPTO, volume2442of LNCS,pages288–303.Springer,2002

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, 17 Mei 2010
ttd



Mochamad Reza Akbar – 13507131