Mutated Hash (MH)

Robbi Kurniawan - 13504015
Jurusan Teknik Informatika ITB, Bandung
email: if14015@students.if.itb.ac.id, robbi.kurniawan@yahoo.com

Abstract – *Hash* merupakan sebuah algoritma yang digunakan untuk membangkitkan nilai karakteristik data yang dapat digunakan untuk menentukan integritas suatu data. Fungsi *hash* yang baik haruslah memenuhi dua syarat: *collision free* dan *one way*. Pada makalah ini, dirancang sebuah algoritma *hash* yang berbasiskan algoritma genetis dan perhitungan matematis sederhana yang dinamakan *Mutated Hash (MH)*. Dari proses pengujian secara *black box* didapatkan hasil bahwasanya *MH* memenuhi sifat *collision free* dan *one way*.

Kata Kunci*: Hash, Genetic Algorithm, Data integrity.*

1. PENDAHULUAN

Salah satu aspek dalam keamanan / kriptografi adalah integritas data. Integritas data dapat dicapai dengan memberikan nilai yang unik untuk setiap data. Nilai unik tersebut dibangkitkan dengan sebuah teknik / fungsi yang disebut sebagai fungsi *hash*. Fungsi *hash* merupakan sebuah fungsi satu arah (*non-reversible* / tidak punya balikan) dengan masukan berupa data berukuran berapa pun dan keluaran berupa nilai unik untuk data tersebut dengan ukuran yang tetap. Apabila data masukan berbeda, maka akan menghasilkan nilai keluaran yang berbeda pula. Nilai keluaran fungsi *hash* sering disebut sebagai nilai *hash*.

Dalam fungsi *hash*, ada dua buah parameter yang menjadi indikator tingkat keamanan fungsi *hash* tersebut [1]. Kedua parameter tersebut adalah:

a. *Collision Free*
   Sebuah fungsi hash dikatakan *collision free* apabila tidak mungkin secara komputasi menemukan dua buah data yang berbeda tetapi memiliki nilai *hash* yang sama.

b. *One Way*
   Sebuah fungsi hash dikatakan *one way* apabila diketahui sebuah nilai *hash*, tidak mungkin secara komputasi menemukan sebuah data yang memiliki nilai *hash* tersebut.

Apabila kedua parameter di atas dipenuhi oleh sebuah fungsi *hash*, maka dikatakan fungsi *hash* tersebut aman secara kriptografi.

Pendekatan yang umumnya dilakukan dalam merancang sebuah fungsi *hash* yang memenuhi kedua syarat di atas adalah dengan menggunakan algortima yang berdasarkan fungsi matematika sederhana dengan penggunaan beberapa konstanta yang dipilih secara khusus. Penggunaan algoritma yang murni berdasarkan fungsi matematika memiliki kelemahan yaitu dapat dirumuskan dengan pasti walaupun membutuhkan analisis yang rumit. Hal ini terbukti dari pecahnya dua buah algoritma *hash* yaitu MD5 dan SHA-1.

Oleh karena itu, dibutuhkan sebuah perancangan algoritma fungsi *hash* yang mengkombinasikan antara fungsi matematika dengan kedinamisan algoritma.
Kedinamisan algoritma di sini berarti apabila terdapat dua buah masukan fungsi yang berbeda, maka algoritma yang digunakan untuk menghasilkan nilai keluaran juga berubah / berbeda. Dengan demikian akan semakin sulit bagi kriptanalis untuk menentukan *behaviour* dari algoritma / fungsi *hash* tersebut.

Pada makalah ini akan dirancang sebuah algoritma / fungsi *hash* yang berdasarkan rumus matematis sederhana dan algoritma yang dinamis. Fungsi *hash* tersebut dinamakan *Mutated Hash (MH)*. Dalam *MH*, digunakan teknik / algoritma genetis dalam mengatur kedinamisan algoritma karena sifatnya yang sangat acak dan dinamis.

2. ANALISIS DAN PERANCANGAN

Sebagaimana fungsi *hash* lainnya, *MH* mendapat masukan berupa data dan menghasilkan sebuah nilai *hash* yang menjadi karakteristik dan unik untuk data tersebut. Perubahan satu bit dalam data masukan akan mempengaruhi secara signifikan nilai *hash* keluaran.

Dalam *MH*, data yang dimasukkan berupa blok-blok berukuran 64 byte atau 512 bit ditambah informasi panjang data sebesar 4 byte.

Apabila suatu data dengan panjang *l* byte ingin dicari nilai *hash*-nya, maka data tersebut akan terlebih dahulu ditambahkan dengan informasi panjang data sehingga total data yang dimasukkan adalah *l+4* byte.

Apabila *l+4* bukan merupakan kelipatan 64 byte, maka bit 0 disisipkan antara data awal dengan informasi panjang data sehingga total data yang dimasukkan merupakan kelipatan 64 byte.

Data yang panjangnya telah merupakan kelipatan 64 byte kemudian dibagi menjadi blok-blok (B_0 ... B_n) dan setiap bloknya dimasukkan ke dalam fungsi *hash MH* secara berurutan (B_0 kemudian B_1, dan seterusnya) hingga seluruh blok dimasukkan ke dalam fungsi *hash MH*. Hasil keluaran fungsi *hash MH* setelah blok terakhir dimasukkan (B_n) menjadi nilai *hash* untuk seluruh data tersebut.

Oleh karena dalam fungsi *hash MH* menggunakan algoritma genetis, maka diperlukan fungsi pembangkit bilangan acak semu dan fungsi penghitung nilai *fitness* untuk setiap individu / kromosom dalam algoritma genetis. Dengan demikian terdapat tiga buah komponen pembentuk *MH*: pembankit bilangan acak semu, penghitung nilai *fitness*, dan algoritma genetis.

2.1. *Pembangkit Bilangan Acak Semu*

Pembangkit bilangan acak semua yang dibutuhkan tidak perlu kuat secara kriptografi karena nilai-nilai yang dibangkitkan hanya digunakan secara internal sehingga. Akan tetapi, sifat *chaos* dibutuhkan sehingga bilangan acak yang dihasilkan tidak memiliki pola. Hal ini sangat penting untuk menjaga kedinamisan fungsi *hash* agar pola algoritmanya tidak dapat ditentukan.

Seluruh fungsi pembangkit bilangan acak semu membutuhkan umpan. Umpan yang digunakan oleh pembangkit bilangan acak semu adalah blok-blok 64 byte yang menjadi masukan fungsi. Dengan demikian, setiap ada masukan data, umpan fungsi pembangkit bilangan acak semu pun ikut berubah.

Oleh karena bilangan acak yang dihasilkan merupakan fungsi dari data masukan, maka data masukan yang berbeda akan menghasilkan nilai bilangan acak yang berbeda pula. Hal ini menambah kedinamisan fungsi *hash MH*.

Dengan kebutuhan di atas, dipilih algoritma pembangkit bilangan acak semu standard Java™ (`java.util.Random`) karena memiliki sifat *chaos* dan cepat walaupun tidak kuat secara kriptografi. Pembangkit bilangan acak semu tersebut memiliki masukan umpan dengan besar 8 byte. Agar dapat memenuhi kebutuhan fungsi *MH*, maka algoritma tersebut diubah sehingga memiliki masukan umpan sebesar 16 byte.

Pengubahan algoritma dilakukan pada proses inisialisasi *buffer* dan proses pembangkitan bilangan acak. Pengubahan proses inisialisasi *buffer* *b* dari umpan *u* dapat dilihat pada rumus 1 dan 2 (u_0 merupakan 8 byte awal *u* dan u_1 merupakan 8 byte akhir *u*).

$$b = (u \oplus 0x5deece66d) \wedge 2^{48}$$

Rumus 1. Proses inisialisasi *buffer* sebelum modifikasi

$$b_0 = (u_0 \oplus 0x5deece66d) \wedge 2^{48}$$
$$b_1 = (u_1 \oplus 0x5deece66d) \wedge 2^{48}$$

Rumus 2. Proses inisialisasi *buffer* setelah modifikasi

Sedangkan pengubahan pada proses pembangkitan bilangan acak dalam dilihat pada rumus 3 dan 4 (*o* merupakan bilangan acak yang dibangkitkan dan *b* adalah *buffer*).

$$b = (b \times 0x5deece66d + 0xb) \wedge 2^{48}$$
$$o = b$$

Rumus 1. Proses pembangkitan bilangan acak sebelum modifikasi

$$b_0 = (b_0 \times 0x5deece66d + 0xb) \wedge 2^{48}$$
$$b_1 = (b_1 \times 0x5deece66d + 0xb) \wedge 2^{48}$$
$$o = b_0 \oplus b_1$$

Rumus 2. Proses pembangkitan bilangan acak setelah modifikasi

2.2. *Penghitung nilai fitness*

Nilai *fitness* merupakan nilai karakteristik sebuah individu / kromosom pada algoritma genetis. Dalam fungsi *hash MH*, digunakan fungsi CRC-32 sebagai penghitung nilai *fitness* suatu individu. Adapun masukan fungsi CRC-32 adalah gen suatu individu.

Dengan menggunakan fungsi CRC-32, maka nilai *fitness* suatu individu akan berkisar antara `0-4,294,967,295`.

Nilai *fitness* suatu individu berubah apabila mengalami perubahan gen dan atau melakukan perkawinan dengan invididu lainnya dan atau mengalami perpindahan tempat. Lebih jelas mengenai perpindahan tempat dapat dilihat pada Bagian 2.3.

2.3. *Algoritma Genetis*

Algoritma genetis merupakan sebuah algoritma yang digunakan untuk mencari kombinasi acak. Setiap kombinasi direpresentasikan sebagai sebuah individu / kromosom sedangkan setiap konfigurasi kombinasi tersebut dipresentasikan sebagai sebuah gen dan setiap individu memiliki 16 buah gen.

Pencarian kombinasi acak dilakukan dengan pembangkitan suatu individu baru berdasarkan individu sebelumnya. Pembangkitan suatu individu baru dilakukan dengan melakukan perkawinan antara dua buah individu yang telah ada sebelumnya.

Oleh karena sifat ini akan membangkitkan individu dengan jumlah yang sangat besar, maka apabila saat individu yang dibangkitkan dan jumlah individu telah mencapai jumlah maximum (1024), maka individu yang paling tua (paling lebih dahulu ada) akan dihilangkan / mati. Dengan aturan ini, maka jumlah individu tidak akan lebih dari 1024 individu.

Apabila individu baru yang dibangkitkan memiliki nilai *fitness* kurang dari atau sama dengan `8,589,934`, maka akan dipilih secara acak dua individu yang akan bertukar tempat satu sama lain. Setelah bertukar tempat, individu tersebut akan mengalami mutasi dengan kemungkinan `100%`.

Algoritma perkawinan antara suatu invididu (kelompok gen *a*) dengan individu lainnya (kelompok gen *b*) yang menghasilkan individu baru (kelompok gen *r*) adalah sebagai berikut:

1. Pilih bilangan acak integer *c* yang nilainya antara `1-16`.
2. Untuk i = *1...c*, $r_i = a_i \times b_i \pmod{2^8}$
3. Untuk i = *c ... 16*, $r_i = a_i + b_i \pmod{2^8}$
4. *Update* nilai *fitness* individu *a* dan *b*.
5. Lakukan mutasi terhadap individu *r* dengan kemungkinan *mrate*.
6. Set nilai *fitness* individu *r* dengan nilai *0*.
7. *Update* nilai *fitness* individu *r*.
8. *mrate = (mrate + (*nilai *fitness r / 4294967295)) / 2*.

Sedangkan algoritma mutasi adalah sebagai berikut:

1. Bangkitkan bilangan acak real *p* antara nilai *0..1*
2. Apabila *p* < kemungkinan, maka lakukan langkah 3 s.d. 5. Apabila tidak, mutasi tidak dilakukan.
3. Bangkitkan bilangan acak integer *i* antara nilai *1..16*
4. Bangkitkan bilangan acak integer *x* antara nilai *0..255*
5. $gen_i = gen_i - x$

2.4. *Algoritma Utama*

Dari analisis di atas, dapat dikonstruksi algoritma utama fungsi *hash MH*:

1. Inisialisasi umpan bilangan acak semu dengan umpan *default* `C>3[4R=e*BG!Q8D$`.
2. Inisialisasi 8 individu awal dengan gen merupakan bilangan acak semu.
3. Inisialisasi *mrate* dengan nilai 0.
4. Setiap blok yang dimasukkan akan diproses dengan algoritma:
    a. Inisialisasi umpan bilangan acak semu dengan umpan *default*.
    b. Bagi blok menjadi 4 buah kelompok gen (b_1, b_2, b_3, b_4) yang masing-masing berukuran 16 byte.
    c. Pilih secara acak satu individu (*i*)
    d. Lakukan perkawinan antara gen *i* dengan gen b_1.
    e. Pilih secara acak satu individu lain (*i*)
    f. Lakukan perkawinan antara gen *i* dengan gen b_2.
    g. Pilih secara acak satu individu lain (*i*)
    h. Lakukan perkawinan antara gen *i* dengan gen b_3.
    i. Pilih secara acak satu individu lain (*i*)
    j. Lakukan perkawinan antara gen *i* dengan gen b_4.
    k. Inisialisasi umpan bilangan acak semu dengan umpan 16 byte pertama blok.
    l. Hitung nilai *rcount = jumlah individu / 10.*
    m. Apabila *rcount < 2* dan *jumlah individu > 0* maka *rcount = 2.*
    n. Lakukan langkah (o) s.d. (p) sebanyak *rcount* kali.
    o. Pilih secara acak dua individu (*i* dan *j*).
    p. Lakukan perkawinan antara *i* dan *j*.
    q. Inisialisasi umpan bilangan acak semu dengan umpan 16 byte kedua blok.
    r. Lakukan langkah (l) s.d. (p).
    s. Inisialisasi umpan bilangan acak semu dengan umpan 16 byte ketiga blok.
    t. Lakukan langkah (l) s.d. (p).
    u. Inisialisasi umpan bilangan acak semu dengan umpan 16 byte keempat blok.
    v. Lakukan langkah (l) s.d. (p).
5. Inisialisasi umpan bilangan acak semu dengan umpan *default*.
6. Lakukan langkah (3.l) s.d. (3.p) sebanyak 5 kali
7. Hitung nilai *hash* (*h*) dengan algoritma ($fitness_i$ adalah nilai *fitness* untuk individu ke-*i*):
    a. *i = 0*
    b. *j = 0*
    c. $h_j = h_j$ *xor* $(fitness_i / 16{,}777{,}216)$
    d. *i = i + 1*
    e. *j = (j + 1) mod 16*
    f. $h_j = h_j$ *xor* $((fitness_i \bmod 16{,}777{,}216) / 65{,}536)$
    g. *i = i + 1*
    h. *j = (j + 1) mod 16*
    i. $h_j = h_j$ *xor* $((fitness_i \bmod 65{,}536) / 256)$
    j. *i = i + 1*
    k. *j = (j + 1) mod 16*
    l. $h_j = h_j$ *xor* $(fitness_i \bmod 256)$
    m. *i = i + 1*
    n. *j = (j + 1) mod 16*
    o. Lakukan langkah (c) s.d. (o) selama *i < jumlah individu*

3. PENGUJIAN

Pada makalah ini dibatasi pengujian dilakukan menggunakan metode non-formal dan secara black-box. Beberapa pengujian yang akan dilakukan:

1. Pengujian dengan algoritma Las Vegas
2. Kecepatan algoritma

Pengujian dilakukan dengan mengimplementasikan algoritma *IKG* dalam bahasa *C++* dan menjalankannya pada sebuah komputer *PC* dengan *processor* berkecepatan 1,6 GHz.

3.1. *Pengujian dengan Algoritma Las Vegas*

Algoritma *las vegas* adalah algoritma yang didesain secara khusus untuk menemukan apakah terdapat dua buah data yang berbeda memiliki nilai *hash* yang sama [1].

Algoritma ini bekerja dengan membangkitkan secara acak satu buah data yang berbeda dengan data asal kemudian menghitung nilai *hash* nya dan memperbandingkan nilai *hash* data pembanding dengan nilai *hash* data yang dibangkitkan. Apabila sama, maka ditemukan *collision*. Apabila tidak sama, maka akan dibangkitkan kembali secara acak satu buah data yang berbeda lagi dan begitu seterusnya.

Setelah diuji dengan membangkitkan 554,200,001 blok data yang berbeda data tetapi dengan ukuran yang sama, didapatkan hasil tidak ada blok data yang memiliki nilai *hash* yang sama.

3.2. *Kecepatan Algoritma*

Perhitungan kecepatan dilakukan dengan dua pendekatan:

1. Perhitungan *m* buah data dengan ukuran masing-masing 1 blok
2. Perhitungan 1 buah data dengan ukuran *n* blok

Dari dua pendekatan tersebut, didapatkan hasil rata-rata yang sangat berbeda. Pendekatan pertama dilakukan dengan membangkitkan 554,200,01 buah data berukuran 1 blok dan memiliki kecepatan rata-rata pembangkitan 1,507,963 byte data per detik.

Sedangkan melalui pendekatan kedua dilakukan dengan membangkitkan 3 buah data dengan ukuran masing-masing 100,000 blok, 500,000 blok dan 1,000,000 blok dan didapat kecepatan rata-rata pembangkitan 109,000 byte data per detik.

4. KESIMPULAN

Fungsi *hash MH* merupakan algoritma *hash* yang memenuhi dua buah syarat amanya sebuah fungsi *hash* yaitu *collision free dan one-way*. Hal ini dikarenakan algoritma yang dipakai sangat dinamis bergantung pada data masukan. Selain itu juga telah dibuktikan melalui pengujian *black-box* dengan menggunakan algoritma *las vegas*.

Walaupun aman, *MH* merupakan algoritma *hash* yang sangat lambat apabila dibandingkan dengan algoritma *hash* lainnya seperti *MD5* dan *SHA*.

5. SARAN

Dilakukan pengujian kekuatan algoritma genetis dengan pengujian *white-box* dan matematika formal.

DAFTAR REFERENSI

[1] Stinson, Douglas. *Cryptography: Teory and Practice*. CRC Press. Maret 1995.

[2] Konar, Amit. *Artificial Intelligence and Soft Computing*. CRC Press. 2002.