KRIPTANALISIS TERHADAP FUNGSI HASH

Hardani Maulana – NIM : 13503077

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
E-mail : if13077@students.if.itb.ac.id

ABSTRAKSI

Makalah yang berjudul *'Kriptanalisis Terhadap Fungsi Hash'* ini membahas mengenai kriptanalisis, atau proses pemecahan kode kriptografi terhadap salah satu fungsi yang cukup penting di bidang kriptografi, fungsi *hash*. Fungsi *hash* banyak digunakan dalam bidang keamanan kriptografi, seperti tanda tangan digital, penggenerasi bilangan acak, pembentukan kunci, otentikasi pesan, pengenalan kode, dan proteksi integrasi data. Ada banyak fungsi *hash* yang telah dikembangkan, di antaranya adalah MD5 dan SHA dengan berbagai tipe variasinya. Namun belakangan di dunia kriptografi SHA-1 adalah fungsi *hash* yang paling banyak digunakan. Namun seiring penggunaan fungsi *hash* dalam pendukung keamanan di kriptografi, berkembang pula ilmu kriptanalisis untuk memecahkan kode pesan, atau *message digest* (red - Dalam makalah ini akan disebut sebagai nilai *hash)* yang dihasilkan oleh algoritma fungsi *hash.* Akibatnya algoritma fungsi *hash* harus terus dikembangkan untuk tingkat keamanan yang lebih baik lagi. Hal ini menimbulkan anggapan bahwa perkembangan algoritma kriptografi didukung dengan berhasilnya dipecahkan sebuah algoritma kriptografi yang lain. Karena itu analisis mengenai kriptanalisis terhadap fungsi *hash* perlu dilakukan untuk mengembangkan sistem kriptografi yang lebih baik lagi.

Fungsi *hash* memiliki dua properti khusus, yaitu satu arah dan resistensi terhadap kolisi. Kriptanalisis pada fungsi *hash* dilakukan dengan dua cara yang membuktikan bahwa dua properti tersebut tidak valid. Cara yang pertama adalah dengan mendapatkan pesan asli dari nilai *hash* yang ada. Karena fungsi *hash* adalah fungsi satu arah, maka seharusnya hal tersebut tidaklah mungkin dapat dilakukan. Bila hal tersebut dapat dilakukan maka fungsi *hash* telah terpecahkan. Sedangkan cara yang kedua adalah dengan menemukan dua pesan yang dapat menghasilkan nilai *hash* yang sama. Bila hal tersebut terjadi, maka pesan asli dapat disamarkan sehingga terjadi kesalahpahaman dalam pengartian pesan. Hal ini sangat berbahaya terutama dalam transaksi uang yang dapat menghasilkan nilai berbeda hanya dengan perbedaan satu digit nol.

Beberapa teknik kriptanalisis yang sudah ada dan terbukti berhasil memecahkan fungsi *hash* adalah *differential cryptanaysis* dan *birthday attack.* Kedua teknik ini sebenarnya menggunakan spesifikasi kedua dari cara melakukan kriptanalisis pada fungsi *hash,* yaitu menemukan dua pesan yang menghasilkan nilai pesan yang sama. Dalam dunia kriptografi cara ini disebut dengan *collision attack* atau serangan kolisi. Dengan melihat kenyataan tersebut seharusnya dapat dianalisis apakah memang hanya serangan kolisi saja yang berpeluang memecahkan sebuah fungsi *hash*, atau masih ada kemungkinan untuk menghasilkan pesan asli dari nilai *hash* yang ada. Sehingga para pengembang fungsi *hash* selanjutnya dapat menetapkan fokus dari fungsi *hash* yang dikembangkan.

Sejalan dengan pernyataan bahwa perkembangan kriptografi akan dipengaruhi oleh pemecahan atau serangan pada algoritma kriptografi tertentu, maka studi mengenai kriptanalisis terhadap fungsi *hash* diharapkan dapat memunculkan ide untuk improvisasi dan perbaikan algoritma fungsi *hash* di kemudian hari. Hal ini tentunya akan sangat signifikan dalam bidang keamanan mengingat perannya yang besar dalam penggunaan di kehidupan teknologi informasi yang pertumbuhannya sangat cepat dari waktu ke waktu.

Kata kunci: fungsi *hash*, kriptanalisis, serangan

1. Pendahuluan

Dalam perkembangan fungsi *hash* yang terjadi pada sepuluh tahun terakhir, banyak sekali kejadian yang menarik menyangkut dengan keamanan dari algoritma fungsi *hash* yang ada. Algoritma MD5, sejak dikembangkan dari MD4 pada tahun 1992 oleh Ron Rivest akhirnya hampir dipecahkan oleh Hans Dobbertin sekitar tahun 1995. Pada tahun tersebut pula NSA (National Security Agency) mengembangkan algoritma SHA dari MD4 dan dianggap sebagai kelanjutan dari pendahulunya, MD5, yang telah digunakan secara luas. Kelebihan dari SHA dibandingkan dengan MD5 adalah pada putarannya yang berjumlah 80 kali dibandingkan dengan MD5 yang hanya empat putaran. SHA disebut aman karena ia dirancang sedemikian rupa sehingga secara komputasi tidak mungkin menemukan pesan yang berkoresponden dengan nilai *hash* yang diberikan. Perkembangan SHA menghasilkan beberapa varian. Namun yang paling sering dipakai adalah SHA-1.

Di konferensi kriptografi pada tahun 1998, A. Joux mempresentasikan *differential attack* dengan peluang 2^{-61}. Pada tahun 2004, Xiaoyun Wang, Yiqun Lisa Yin, dan Hongbo Yu mempublikasikan metode serangan yang dapat digunakan untuk memecahkan fungsi *hash* yang telah diketahui lemah, yaitu MD4, MD5, dan SHA-0. Dan berikutnya pada Februari 2005, hasil yang serupa memiliki efek pada SHA-1, fungsi *hash* yang diyakini dapat mencegah serangan baru tersebut. Selanjutnya banyak lagi keraguan muncul setelah pembahasan mengenai kemungkinan multikolisi dari iterasi fungsi *hash* pada tahun berikutnya.

Semakin menambah keyakinan dalam keberhasilan kriptanalisis, pada tanggal 1 Maret 2005, Arjen Lenstra, Xiaoyun Wang, dan Benne de Weger mendemonstrasikan pembentukan dua buah sertifikat X.509 dengan kunci publik yang berbeda tetapi mempunyai nilai hash yang sama. Dan beberapa hari kemudian Vlastimil Klima memperbaiki algoritma Lenstra dkk yang dapat menghasilkan kolisi MD5 hanya dalam waktu beberapa jam dengan menggunakan komputer PC.

Akhirnya bersaranglah pertanyaan di otak para pakar teknologi informasi di dunia, terutama yang konsen di bidang keamanan, perlukah mengganti algoritma yang ada bila sebelumnya desain keamanan yang digunakan mengacu pada MD4 atau MD5 (yang diketahui lemah dari tahun 1995) ataupun SHA-1 (yang dipercayai kuat hingga Februari 2005).

NIST (National Institute of Standards and Technology) memberikan rekomendasi agar penggunaan algoritma MD5, dan yang artinya termasuk juga MD4 dihentikan. Apalagi dengan perkembangan kemampuan pemrosesan computer yang semakin cepat, maka akan lebih aman jika menggunakan SHA-1 hingga tahun 2010. Setelah 2010, NIST merencanakan untuk mengakhiri penggunaan SHA-1 dengan mengembangkan SHA-256.

Melihat perkembangan yang cepat dalam 'dunia *hash*', di mana pembentukan algoritma baru dan pemecahan dari algoritma yang sudah ada saling adu kecepatan, maka sudah sewajarnya improvisasi terhadap algoritma yang sudah ada dilakukan setiap saat.

2. Fungsi Hash

Sebelum membahas mengenai kriptanalisisnya, ada baiknya dibahas dahulu mengenai fungsi *hash* itu sendiri. Bab ini akan membahas mengenai deskripsi fungsi *hash* secara umum dan beberapa tipe fungsi *hash* yang umum digunakan. Termasuk juga algoritma dan fungsi yang digunakan dalam fungsi *hash*.

2.1. Deskripsi Fungsi Hash

Fungsi *hash* adalah fungsi yang menerima masukan *string* yang panjangnya sembarang lalu mentransformasikannya menjadi *string* keluaran yang panjangnya tetap. Pada umumnya panjang *string* keluaran ini ukurannya jauh lebih kecil daripada ukuran *string* semula.

Struktur fungsi *hash* pada umumnya adalah sebagai berikut:

1. Menerima *string* masukan yang panjang, lalu membaginya menjadi beberapa blok:

   $M_1, M_2, M_3, \ldots M_5$ (pada blok terakhir dilakukan *padding*)

2. Misalkan H adalah keadaan nilai dari n bit, dan f adalah fungsi yang beroperasi pada blok dan nilai *hash*, maka:

   H_0 = Vektor Inisialisasi

   $H_i = f(H_{i-1}, M_i)$ untuk i = 1,2,3, … s

3. Keluaran terakhir dari fungsi adalah nilai *hash* dengan panjang n bit.

Fungsi *hash* memiliki dua atribut, yaitu resistensi terhadap kolisi dan satu arah. Resistensi terhadap kolisi artinya tidak dapat ditemukan dua pesan dengan nilai *hash* yang sama. Sedangkan satu arah berarti pesan yang sudah diubah menjadi *message digest* tidak dapat dikembalikan lagi menjadi pesan semula *(irreversible*).

Beberapa aplikasi dari fungsi *hash* adalah sebagai berikut:

1. Menjaga integritas data
2. Menghemat waktu pengiriman
3. Menormalkan panjang data yang beraneka ragam

Peta kekuatan fungsi *hash* yang ada saat ini yang dikeluarkan oleh NIST adalah sebagai berikut:

Size in bits

Sym. Key	56	80	112	128	192	256
Hash (for signatures)	128	160	224	256	384	512
Pub. Key	512	1k	2k	3k	7.5k	15k
EC	160		224	256	384	512

Sym. Key: Symmetric key encryption algorithms
MAC: Message Authentication Code
Pub. Key: Factoring or discrete log based public key algorithms
EC: Elliptic Curve based public key algorithms
White background: expected to be secure until at least 2030
Yellow background: Phase out use by 2010
Black background: not secure now

NIST National Institute of Standards and Technology

2.2. MD4

Algoritma MD4 melakukan kompresi pesan menjadi nilai *hash* dalam panjang 128 bit. Pada setiap pesan yang diberikan, pertama algoritma akan melakukan *padding*, penambahan pesan pengganjal, menjadi pesan dengan panjang kelipatan dari 512 bit.

Selanjutnya untuk setiap 512 bit blok pesan, MD4 mengkompresinya menjadi 128 bit nilai *hash* menggunakan fungsi *hash*. Fungsi MD4 memiliki tiga putaran. Setiap putaran menggunakan fungsi boolean yang berbeda seperti yang didefinisikan berikut ini:

$F(X, Y, Z) = (X \wedge Y) \vee (\neg X \wedge Z)$

$G(X, Y, Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$

$H(X, Y, Z) = X \oplus Y \oplus Z$

Pada operasi ini X, Y dan Z memiliki ukuran 32 bit. Operasi pada ketiga fungsi dijalankan dalam bentuk bit. ¬X merupakan komplemen dari X, ^, $\oplus$ dan v menunjukkan operasi bit AND, XOR dan OR.

Setiap putaran mengulang 16 langkah operasi yang serupa, dan pada setiap langkah empat variabel yang terkait, a, b, c, dan d diperbaharui.

$\varphi_0(a,b,c,d,m_k,s)=((a+F(b,c,d)+m_k) \bmod 2^{32})<<<s$

$\varphi_1(a,b,c,d,m_k,s)=((a+G(b,c,d)+m_k+0x5a827999) \bmod 2^{32})<<<s$

$\varphi_2(a,b,c,d,m_k,s)=((a+H(b,c,d)+m_k+0x6ed9eba1) \bmod 2^{32})<<<s$

Nilai inisial dari MD4 didefinisikan sebagai berikut:

(a, b, c, d) = (0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476)

Fungsi *hash* MD4: Untuk setiap 512 bit blok M fungsi didefinisikan sebagai berikut:

1. Untuk (aa, bb, cc, dd) variabel maukan untuk blok M. Bila M adalah blok pertama yang dimasukkan, maka nilai (aa, bb, cc, dd) adalah nilai vektor inisialisasi. Selain itu maka (aa, bb, cc,

dd) merupakan keluaran dari blok sebelumnya.

2. Lakukan 48 langkah berikut dalam tiga putaran:

   Untuk j = 0, 1, 2 dan i = 0, 1, 2, 3

   $a = \varphi_j (a, b, c, d, w_{j,4i}, s_{j,4i})$

   $d = \varphi_j (d, a, b, c, w_{j,4i+1}, s_{j,4i+1})$

   $c = \varphi_j (c, d, a, b, w_{j,4i+2}, s_{j,4i+2})$

   $b = \varphi_j (b, c, d, a, w_{j,4i+3}, s_{j,4i+3})$

   Di sini $s_{j,4i+k}$ (k = 0, 1, 2, 3) adalah langkah konstan yang tidak bergantung satu sama lain, $w_j,4_{i+k}$ adalah isi pesan dan <<< $s_{j,4i+k}$ adalah *circular left-shift* dari posisi bit $s_{j,4i+k}$.

3. Tambahkan variabel a, b, c, dan d ke dalam variabel aa, bb, cc, dan dd untuk memproduksi nilai variabel akhir dari masukan:

   aa = (a + aa) mod 232

   bb = (b + bb) mod 232

   cc = (c + cc) mod 232

   dd = (d + dd) mod 232

   Jika M adalah blok terakhir dari pesan, maka H(M) = aa|bb|cc|dd adalah nilai *hash* dari pesan M. Bila bukan merupakan blok terakhir maka ulangi proses di atas kepada 512 bit blok pesan berikutnya dan (aa, bb, cc dd) sebagai variabel masukan.

2.3. RIPEMD

RIPEMD menggunakan putaran fungsi yang serupa dengan MD4 dalam enam fungsi operasi sebagai berikut:

$\varphi_0(a,b,c,d,m_k,s)=((a+F(b,c,d)+m_k) \bmod 2^{32})<<<s$

$\varphi_1(a,b,c,d,m_k,s)=((a+G(b,c,d)+m_k+0x5a827999) \bmod 2^{32})<<<s$

$\varphi_2(a,b,c,d,m_k,s)=((a+H(b,c,d)+m_k+0x6ed9eba1) \bmod 2^{32})<<<s$

$\psi_0(a,b,c,d,m_k,s)=((a+F(b,c,d)+m_k+0x50a28be6) \bmod 2^{32})<<<s$

$\psi_1(a,b,c,d,m_k,s)=((a+G(b,c,d)+m_k) \bmod 2^{32})<<<s$

$\psi_2(a,b,c,d,m_k,s)=((a+H(b,c,d)+m_k+0x5c4dd124) \bmod 2^{32})<<<s$

Untuk memudahkan pendeskripsian fungsi RIPEMD, fungsi MD4 φ0, φ1 dan φ2 dituliskan dengan MD4(φ0, φ1, φ2,M).

Fungsi *hash* RIPEMD: Menggunakan dua fungsi MD4, fungsi sebelah kiri adalah MD4($\varphi_0,\varphi_1,\varphi_2$,M), dan fungsi sebelah kanan adalah MD4(ψ_0,ψ_1,ψ_2,M). Kedua fungsi memiliki nilai inisial yang sama seperti MD4.

1. Untuk (a, b, c, d) variabel masukan untuk blok M seperti pada MD4.
2. Lakukan dua fungsi operasi dari MD4

   (aa,dd,cc,bb,) ← MD4($\varphi_0,\varphi_1,\varphi_2$,M),

   (aaa,ddd,ccc,bbb)← D4(ψ_0,ψ_1,ψ_2,M).

3. Hasil keluaran (a, b, c, d) untuk hasil operasi fungsi pada blok M adalah sebagai berikut:

   $a = (b + cc + ddd) \bmod 2^{32}$

   $b = (c + dd + aaa) \bmod 2^{32}$

   $c = (d + aa + bbb) \bmod 2^{32}$

   $d = (a + bb + ccc) \bmod 2^{32}$

2.4. MD5

Algoritma MD5 menerima masukan berupa pesan dengan ukuran sembarang dan menghasilkan message digest yang panjangnya 128 bit.

Langkah-langkah pembuatan *message digest* secara dimulai dengan penambahan bit-bit pengganjal (*padding bits*). Pesan ditambah

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

Selanjutnya dilakukan penambahan nilai panjang pesan semula. Pesan yang telah diberi bit-bit pengganjal selanjutnya ditambah lagi dengan 64 bit yang menyatakan panjang pesan semula. Jika panjang pesan > 264 maka yang diambil adalah panjangnya dalam modulo 264. Dengan kata lain, jika panjang pesan semula adalah *K* bit, maka 64 bit yang ditambahkan menyatakan *K* modulo 264. Setelah ditambah dengan 64 bit, panjang pesan sekarang menjadi kelipatan 512 bit.

MD5 membutuhkan 4 buah penyangga (buffer) yang masing-masing panjangnya 32 bit. Total panjang penyangga adalah 4x32 = 128 bit. Keempat penyangga ini menampung hasil antara dan hasil akhir. Keempat penyangga ini diberi nama A, B, C, dan D. Setiap penyangga diinisialisasi dengan nilai-nilai (dalam notasi HEX) sebagai berikut:

(A, B, C, D) = (01234567, 89ABCDEF, FEDCBA98, 76543210)

Pengolahan pesan dalam blok berukuran 512 bit dilakukan dengan langkah-langkah sebagai berikut:

1. Pesan dibagi menjadi L buah blok yang masing-masing panjangnya 512 bit (Y_0 sampai Y_{L-1}).
2. Setiap blok 512-bit diproses bersama dengan penyangga MD menjadi keluaran 128-bit. Proses ini terdiri dari 4 buah putaran, dan masing-masing putaran melakukan operasi dasar MD5 sebanyak 16 kali dan setiap operasi dasar merupakan operasi yang berbeda.

   a ← b + CLSs(a + g(b, c, d) + X[k] + T[i])

   a,b,c,d = empat peubah penyangga (berisi nilai penyangga A, B, C, D)

   CLSs = *circular left shift* sebanyak s bit

   X[k] = kelompok 32-bit ke-k dari blok 512 bit pesan ke-q. Nilai k = 0 sampai 15.

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

   + = operasi penjumlahan modulo 232

   g = salah satu fungsi berikut:

   F(b, c, d) = (b ^ c) v (¬b ^ d)

   G(b, c, d) = (b ^ d) v (c ^ ¬d)

   H(b, c, d) = b ⊕ c ⊕ d

   I(b, c, d) = c ⊕ (b ^ ¬d)

   ¬X merupakan komplemen dari X, ^, ⊕ dan v menunjukkan operasi bit AND, XOR dan OR.

3. Setelah putaran keempat, a, b, c, dan d ditambahkan ke A, B, C, dan D, dan selanjutnya algoritma memproses untuk blok data berikutnya.
4. Keluaran akhir dari algoritma *MD5* adalah hasil penyambungan bit-bit di *A*, *B*, *C*, dan *D*.

2.5. SHA-1

SHA-1 menerima masukan berupa pesan dengan ukuram maksimum 264 bit (2.147.483.648 gigabyte) dan menghasilkan message digest yang panjangnya 160 bit, lebih panjang dari message digest yang dihasilkan MD5 yang hanya 128 bit. Gambaran umum pembuatan message digest dengan menggunakan algoritma ditunjukkan pada gambar berikut ini.

Langkah pembuatan *message digest* dengan SHA-1 hampir serupa dengan MD5, namun memiliki perbedaan pada operasi dan jumlah putarannya.

Langkah awal dilakukan dengan penambahan bit-bit pengganjal (padding bits) dan diikuti dengan penambahan nilai panjang pesan semula. Inisialisasi penyangga (buffer) MD.

Berbeda dengan MD5, SHA membutuhkan lima buah penyangga (*buffer*) yang masing-masing panjangnya 32 bit. Total panjang penyangga adalah 5x32 = 160 bit. Keempat penyangga ini menampung hasil antara dan hasil akhir. Kelima penyangga ini diberi nama A, B, C, D, dan E. Setiap penyangga diinisialisasi dengan nilai-nilai (dalam notasi HEX) sebagai berikut:

(A, B, C, D, E) = (67452301, EFCDAB89, 98BADCFE, 10325476, C3D2E1F0)

Pengolahan Pesan dalam Blok Berukuran 512 bit adalah sebagai berikut:

1. Pesan dibagi menjadi L buah blok yang masing-masing panjangnya 512 bit (Y_0 sampai Y_{L-1}).
2. Setiap blok 512-bit diproses bersama dengan penyangga MD menjadi keluaran 128-bit. Proses ini terdiri dari 80 buah putaran (MD5 hanya 4 putaran), dan masing-masing putaran menggunakan bilangan penambah Kt, yaitu:

   Putaran $0 \leq t \leq 19$ Kt = 5A827999

   Putaran $20 \leq t \leq 39$ Kt = 6ED9EBA1

   Putaran $40 \leq t \leq 59$ Kt = 8F1BBCDC

   Putaran $60 \leq t \leq 79$ Kt = CA62C1D6

3. Setiap putaran menggunakan operasi dasar yang sama (dinyatakan sebagai fungsi f). Operasi dasar SHA dapat ditulis dengan persamaan sebagai berikut:

   $a,b,c,d,e \leftarrow (CLSs(a)+f_t(b,c,d)+e+W_t+K_t)$, a, CLS30(b), c, d

   a,b,c,d,e = lima buah peubah penyangga 32-bit (berisi nilai penyangga A, B, C, D, E)

   t = putaran, $0 \leq t \leq 79$

   CLSs = *circular left shift* sebanyak s bit

   K_t = konstanta penambah

   + = operasi penjumlahan modulo 2^{32}

   f_t = fungsi logika sebagai berikut:

   putaran 0..19 $(b \wedge c) \vee (\neg b \wedge d)$

   putaran 20..39 $b \oplus c \oplus d$

   putaran 40..59 $(b \wedge c) \vee (b \wedge d) \vee (c \wedge d)$

   putaran 60..79 $b \oplus c \oplus d$

   W_t = *word* 32-bit yang diturunkan dari blok 512 bit yang sedang diproses

   Nilai W_1 sampai W_{16} berasal dari 16 *word* pada blok yang sedang diproses, sedangkan nilai W_t berikutnya didapatkan dari persamaan

   $$W_t = W_{t-16} \oplus W_{t-14} \oplus W_{t-8} \oplus W_{t-3}$$

   Setelah putaran ke-79, a, b, c, d, dan e ditambahkan ke A, B, C, D, dan E dan selanjutnya algoritma memproses untuk blok data berikutnya. Keluaran akhir dari algoritma SHA adalah hasil penyambungan bit-bit di A, B, C, D, dan E.

3. Kriptanalisis Fungsi Hash

Sejak lahirnya fungsi *hash* maka muncul pula usaha untuk memecahkannya. Hal ini tidak terlepas dari sifat manusia yang selalu ingin tahu dan tidak mau kalah. Telah banyak serangan-serangan terhadap fungsi *hash* yang berhasil dan mengakibatkan perkembangan fungsi *hash* yang lebih kuat lagi. Namun sejauh ini ujung tombak dari serangan terhadap fungsi *hash* adalah sebuah senjata bernama serangan kolisi yang tak jemu mencari celah kolisi dari sebuah fungsi *hash.*

3.1. Kolisi Pada Fungsi Hash

Aplikasi yang digunakan secara luas yang mengimplementasikan fungsi *hash* adalah sertifikat digital. Sebuah sertifikat digital mengandung nilai *hash* H(b) dari *string* bit b. Untuk memanipulasi sertifikat tersebut maka kriptanalis harus menemukan *string* bit b' yang berbeda dengan b, namun memiliki nilai *hash*

yang sama dengan b. Diberikan H(b),dan mungkin juga isi dari b itu sendiri, menemukan b' yang berbeda dengan nilai *hash* yang berkolisi (H(b) = H(b')) masih dipercayai cukup sulit untuk fungsi *hash* yang biasa digunakan.

Fungsi *hash* dengan nilai n bit memang seharusnya didesain sehingga untuk menemukan pesan yang memiliki nilai *hash* dengan kolisi terhadap pesan aslinya (disebut juga *second pre-images*) membutuhkan 2^n operasi.

Namun untuk kasus fungsi *hash* yang melakukan proses dengan iteratif hal tersebut tidak berlaku. Dan memang fungsi *hash* pada umumnya seperti MD4, MD5, dan seluruh varian SHA menggunakan iteratif.

Untuk fungsi seperti ini, *second pre-images* dapat dikomputasi 2^k kali lebih cepat, untuk nilai $k \geq 0$ yang tidak jauh lebih besar dibandingkan panjang *hash* n. Misalnya $k \leq 55$ untuk SHA-1. Namun, komputasi membutuhkan memori yang proporsional untuk 2^k. Tampaknya hal ini masih sulit untuk dilakukan.

Algoritma fungsi *hash* MD4, MD5, dan SHA telah diketahui selama beberapa tahun. Hal ini tidak pernah menjadi sebuah perhatian. Sehingga fungsi *hash* tersebut masih dipercaya untuk digunakan sebagai proteksi keamanan melawan percobaan penyerangan atau pencurian data inforamsi yang membutuhkan *second pre-images* untuk sebuah nilai *hash* yang ada.

Namun, di balik keamanan yang tercipta lewat fungsi *hash* sebenanya masih ada celah yang dapat ditembus oleh kriptanalis. Hal ini dikarenakan kebebasan yang dimiliki oleh kriptanalis dalam memanipulasi pesan maupun nilai *hash*. Ketika kriptanalis dapat menemukan kolisi dari nilai *hash* pesan asli maka akan sangat berbahaya. Contohnya adalah ketika dokumen digital telah di tandatangani oleh pengirim pesan dengan nilai *hash* H(d) dari dokumen d (sebuah bitstring). Maka bila kriptanalis dapat membuat pesan d' yang memiliki nilai H(d') = H(d), dengan perbedaan isi pada sebagian pesan d. Selanjutnya kriptanalis dapat mengklaim bahwa dokumen d' yang telah ditandatangani. Karena H(d) = H(d') maka klaim dari penyerang tandatangan digital tidak dapat dibantah. Karena itulah fungsi *hash* haruslah memiliki resistensi terhadap kolisi: tidak mungkin secara komputasi dapat dibentuk pesan yang berbeda namun memiliki nilai *hash* yang sama. Hal ini membutuhkan desain dan konstruksi yang baik dari fungsi *hash.*

Karena probabilitas ini jauh lebih besar dari pada yang dibayangkan oleh orang, maka muncul sebuah paradoks. Paradoks ini disebut dengan *birthday paradox* yang memberikan ide pada serangan *birthday attack.* Mengenai *birthday attack* ini akan dibahas kemudian pada sub bab berikutnya.

Hal ini menunjukkan fakta yang general terjadi pada seluruh *fungsi*,baik yang sudah lampau, saat ini, maupun fungsi *hash* masa depan. Hasil kriptanalisis menunjukkan hasil yang lebih buruk pada situasi kolisi antar nilai *hash.* Dapat ditunjukkan bahwa kolisi acak dapat ditemukan dengan tangan untuk MD4 dan dikomputasi dalam hitungan menit untuk MD5. Dengan n = 128, keduanya didesain untuk mencegah 2^{64} usaha menemukan kolisi. Untuk SHA-0 dengan n = 160, kolisi dapat ditemukan dengan usaha kurang dari 2^{39}, jauh lebih sedikit dari usaha membuatnya 2^{80}. Dan yang terakhir, untuk SHA-1, juga dengan n = 160, kolisi dapat ditemukan dengan usaha 2^{69}, lebih kecil dari usaha membuat dengan 2^{80}.

Untuk aplikasi yang mempraktekan MD4, MD5, dan SHA-0 sudah seharusnya tidak lagi digunakan dalam lingkungan di mana para penyerang fungsi *hash* dapat dengan bebas berkeliaran dan memilih nilai *hash* mana yang akan diserang. Ini adalah kasus bahkan bila nilai yang akan di-*hash* harus menemui kebutuhan structural *string*, seperti nilai yang dikandung dalam sertifikat digital, karena kolisi *hash* yang telah dibentuk memenuhi syarat yang diberikan. Di sisi lain, karena MD4 dan MD5 telah dikenal lemah untuk waktu yang lama, dan SHA-0 tidak sering digunakan, maka akibat dari penemuan baru dalam kriptanalisis harus dibatasi. Dari pada itu, komunitas kriptografi tertarik untuk melihat seberapa lemah algoritma MD4, MD5, dan SHA-0 telah berubah dalam bayangan orang banyak. Setelah serangan ini diumumkan, NIST menulis bahwa "SHA-1 tidak lah pecah" dan "Tidak banyak alasan untuk mengharapkan hal itu akan terjadi dalam waktu dekat". Hal ini dapat dilihat bila 2^{69} serangan kolisi acak kepada SHA-1 telah menjadi perhatian penting.

Tapi dalam pengembangan yang mengambil tempat pada akhir 2004 itu sulit dibayangkan hal

itu dating sebagai kejutan yang nyata. SHA-1 dengan ekstensi SHA-I untuk i = 224, 256, 384, dan 512, sejauh ini diketahui tidak terefeksi dengan metode serangan yang terbaru sekalipun.

3.2. *Differential attack*

Differential attack adalah Teknik kriptanalisis yang cukup kuat. Aplikasinya pada blok *cipher* dideskripsikan oleh E. Biham dan A. Shamir dalam bukunya di tahun 1993. *Differential attack* dapat diaplikasikan pada fungsi *hash* oleh Vincent Rijmen dan Bart Preneel dengan cara yang sama dengan aplikasinya pada blok *cipher,* namun dengan beberapa perbedaan sebagai berikut:

1. Pada kasus serangan kolisi, masukan plainteks dimanipulasi. Hal ini membuat serangan pada fungsi *hash* jadi memungkinkan untuk dilakukan. *Differential attack* pada blok *cipher* digunakan sebagai alat enkripsi.
2. Kunci diketahui, dan terkadang kunci dapat dipilih atau alternatif terbaik dari kumpulan kunci yang mungkin. Hal ini dapat dilakukan dalam beberapa cara. Pertama, ketika mencari kolisi, dapat dipilih nilai masukan yang mengikuti karakteristik dengan kemungkinan satu pada putaran tersebut. Kedua peluang dari beberapa karakteristik tergantung keoada kunci. Bila control terhadap kunci dimiliki, maka kunci dipilih secara optimal untuk karakteristiknya. Bila tidak karakteristik dapat dipilih dengan kemungkinan paling optimal dengan kunci yang dimiliki. Ketiga, strategi pembatalan dapat digunakan. Ketika melihat pasangan yang salah, maka secepatnya langsung diabaikan. Untuk kebanyakan masukan dibutuhkan komputasi hanya dalam putaran dengan jumlah yang sedikit.
3. Dalam karakteristik terdapat beberapa batasan: pada analisis blok *cipher* hanya dibutuhkan keluaran karakteristik yang paling berpeluang. Namun untuk fungsi *hash* dibutuhkan karakteristik untuk menciptakan kolisi. Misalnya hasil XOR dari fungsi f harus 0. Artinya keluaran XOR dari blok *cipher* harus sesuai dengan masukan XOR. Selanjutnya, karakteristik harus meliputi seluruh putaran: serangan hanya pada salah satu putaran tidak dapat digunakan. Hal ini mereduksi peluang dari karakteristik.
4. Hanya dibutuhkan satu pasangan kanan untuk menemukan kolisi atau pengembalian pesan semula. Namun yang saat ini dapat dilakukan barulah serangan kolisi.

Dalam melakukan serangan kolisi, dapat dipilih nilai masukan (atau pesan) secara acak saja. Kebebasan ini digunakan untuk meningkatkan peluang untuk mencapai keberhasilan. Tujuan utamanya adalah untuk memilih pesan yang akan menuju kepada karakteristik pada dua putaran pertama dengan probabilitas satu. Selanjutnya akan dijelaskan bagaimana algoritma yang diajukan oleh Vincent Rijmen dan Bart Preneel yang dapat melewati empat putaran dengan probabilitas satu. Dengan tambahan algoritma yang sangat sederhana pun memungkinkan untuk dapat melewati lima putaran.

Langkah pertama: Kalkulasi tabel T1 yang menjabarkan semua nilai dari R1 yang mengikuti karakteristik pada putaran pertama. Berlaku juga untuk tabel T2 dan tabel T3. Karena setiap putaran hanya sedikit S-box yang aktif, makan tabel dapar direduksi dengan menggunakan "*don't cares*" yaitu kita tidak mengindahkan (*don't care*) masukan dari S-box yang tidak aktif dan juga tidak mendefinisikan nilainya.

Langkah kedua: Cocokkan tiga tabel ini dan lihat seluruh nilai yang memungkinkan dari (R1, F2, R3).

Langkah ketiga: Kalkulasi tabel T4 dengan seluruh nilai dari R4. Untuk setiap (R1, F2, R3) balikan (*invert*) putaran kedua dan coba cocokkan nilai yang mungkin dari R2, F(R3) dan R4.

Langkah keempat: Untuk setiap kecocokan yang ditemukan, kalkulasi masukan ke putaran pertama. Dengan mengkalkulasi tabel ekstra dapat memperkirakan empat putaran ini dengan putaran ekstra sebelum putaran pertama.

Sebelumnya telah dilakukan observasi bahwa ada banyak permasalahan untuk menemukan putaran penuh karakteristik dari fungsi *hash*. Satu putaran karakter iteratif dapat memiliki bilangan putaran yang bebas, tapi memiliki kemungkinan yang sangat kecil, yaitu 1:234. Dua

putaran karakter iteratif memiliki kemungkinan yang paling tinggi untuk tujuh putaran atau lebih. Tapi untuk fungsi *hash* hanya bisa diaplikasikan pada varian DES dengan putaran berjumlah ganjil. Hal ini dikarenakan untuk setiap $0 \leftarrow \chi$ putaran memiliki rata-rata peluang 1:234. Tergantung pada kunci putaran, peluang ini dapat menjadi 1:146 atau 1:585. Untuk 13 putaran, serangan membutuhkan bilangan yang sama untuk enkripsi dengan serangan kolisi *brute force* pada *birthday paradox* ($\approx 2^{325}$). Namun karena DES memiliki 16 putaran, maka serangan yang dilakukan membutuhkan karakteristik dengan putaran berjumlah genap. Penggambaran dari satu putaran dan dua putaran karakter iteratif adalah sebagai berikut:

$(\chi , \chi) \quad (\chi , 0)$

$0 \leftarrow \chi \quad 0 \leftarrow 0$

$0 \leftarrow \chi \quad 0 \leftarrow \chi$

$0 \leftarrow \chi \quad 0 \leftarrow 0$

$0 \leftarrow \chi \quad 0 \leftarrow \chi$

$0 \leftarrow \chi \quad 0 \leftarrow 0$

$(\chi , \chi) \quad (\chi , 0)$

Pada bukunya "*Differential cryptanalysis of hash functions based on block ci-phers"*, B. Preneel mengajukan untuk melakukan pencarian masukan nilai χ dengan nilai yang pasti ($\chi \leftarrow \chi$) dan pembangunan blok yang baik untuk karakter iteratif ($0 \leftarrow \chi$). Pada "Block ciphers – analysis, design and applications", L. Knudsen menunjukkan bahwa karakteristik tidak dapat memiliki peluang yang besar. Permasalahannya adalah semua χ dengan peluang yang baik untuk $0 \leftarrow \chi$ memiliki peluang yang kecil untuk $\chi \leftarrow \chi$, dan sebaliknya. L. Knudsen lalu mengajukan untuk menggunakan karakter iteratif dari empat putaran spesial pembangunan blok. Pembuatan blok ini memiliki probabilitas rata-rata $2^{-23.6}$ untuk seluruh kunci, dan $2^{-23.0}$ untuk kunci optimal. Untuk varian DES dengan delapan putaran kompleksitas kerja dapat dibandingkan dengan pencarian kolisi menggunakan *brute force.* Karakter iteratif empat putaran dari L. Knudsen digambarkan sebagai berikut:

(E0000004x , E0000004x)

00000002x ← E0000004x

00000002x ← E0000006x

00000002x ← E0000006x

00000002x ← E0000004x

(E0000004x , E0000004x)

Ambil χ dengan peluang yang baik untuk $0 \leftarrow \chi$. Dibandingkan memasukkan satu putaran $\chi \leftarrow \chi$, dimasukkan lima putaran transisi. Lima putaran ini memiliki probabilitas kecil namun lebih baik dari konstruksi nilai yang tetap. Hal ini bukan merupakan masalah karena nilai masukan yang dipilih untuk putaran transisi memiliki probabilitas satu. Pencarian komputer mengindikasikan bahwa putaran transisi terbaik memiliki bentuk yang tidak simetris. Untuk setiap semua χ yang mungkin nilai α dan β dapat ditemukan. Kombinasi terbaik adalah χ=00196000 dan α=β=04450180. Peluang dari putaran yang berbeda diberikan berikut ini. Tandai bahwa ada nilai χ yang menghasilkan peluang lebih kecil untuk nilai optimal α dan β yang berbeda.

Structure	probability	comments
$0 \leftarrow \chi$	2^{-8}	kunci independen
$\alpha \leftarrow \chi$	$2^{-10.8}$	$2^{-9.95}$ untuk 50% kunci
$\chi \square \alpha \leftarrow \alpha$	$2^{-20.5}$	$2^{-18.1}$ untuk 4.7% kunci

Fakta bahwa peluang dari putaran tergantung dari kunci, dapat digunakan untuk mereduksi rasio pekerjaan: kunci yang memberikan probabilitas kecil atau bahkan nol dapat dieliminasi Kunci yang memberikan nilai peluang bukan nol disebut mendekati optimal (*near-optimal*). Untuk pilihan χ dan α, 4.5% dari kunci adalah mendekati optimal. Kriteria yang lebih kuat untuk mendekati optimal adalah mungkin. Berikut ini digambarkan teori probabilitas dan kompleksitas dari varian DES dengan jumlah putaran yang beragam:

.# kompleksitas peluang putaran

(log2) (log2)

8	-65.8	8
9	-28.8	5
12	-81.8	21.4
13	-43.2	18.9
14	-89.8	29.2
15	-50.3	25.9
16	-97.8	37.0

Probabilitas dari karakteristik diberikan untuk kunci yang mendekati optimal. Kompleksitas dihitung dengan cara menghitung peluang dari putaran di mana nilai masukan yang dikalikan factor reduksi tidak dipilih. Jumlah varian DES dengan jumlah putaran ganjil diterima dengan memilih nilai masukan untuk lima putaran yang diambil secara acak. Berikut ini alternatif dari bagian dari putaran transisi karakter iteratif:

(χ , 0)	(χ, 0)
0 ← 0	0← 0
0 ← χ	0← χ
. . . . .	. . . . .
χ ← χ	α← χ
0 ← χ	χ □ β ← α
. . . . .	χ □ α ← β
	β← χ
	0← 0
(χ , 0)	(χ, 0)

Dengan karakteristik putaran berjumlah genap, fakta bahwa lima putaran dapat dilewati tidak digunakan. Hal ini dapat dilihat pada gambar di atas bahwa empat putaran di tengah dengan probabilitas yang paling kecil di batasi oleh dua putaran dengan probabilitas satu. Tidak ada yang bisa didapatkan dari mengkalkulasi dua putaran ini. Mungkin saja ada karakteristik yang baik dari bentuk lima putaran optimal – lima putaran transisi – enam putaran karakteristik optimal.

3.3. *Birthday Attack*

Birthday attack bukanlah teknik menggunakan tanggal ulangtahun sebagai kode angka rahasia, namun merupakan serangan yang mengimplementasikan kasus *"birthday paradox"*, yaitu kasus di mana ada lebih dari satu orang yang memiliki hari ulang tahun yang sama. Probabilitas dari katakanlah sebuah kelompok dengan anggota 23 orang yang dipilih secara acak, maka peluang setidaknya ada dua orang yang memiliki hari ulangtahun yang sama lebih dari 50 persen. Karena peluang ini lebih besar dari yang diperkirakan oleh sebagian besar orang, maka disebut sebagai paradoks (walaupun hal ini bukanlah suatu paradoks sebenarnya). Hal ini merupakan sebuah fakta yang dapat dibuktikan secara matematis dan diverifikasi secara empirik. Misalkan elemen dipilih secara acak dari koleksi N objek dengan perulangan, maka jumlah yang diperkirakan akan terpilih sebelum elemen tersebut terpilih dua kali adalah sekitar $1.26\sqrt{N}$. Dalam kasus ulang tahun maka N akan bernilai 365 (banyak hari dalam setahun). Maka dalam konteks fungsi *hash* n bit maka N=2n objek yang berbeda. Setelah melakukan fungsi *hash* sebanyak $1.26x2^{n/2}$ *string* bit yang dipilih secara acak maka dapat diperkirakan dua *string* dapat memiliki nilai *hash* yang sama. Hal inilah yang mendasari serangan dengan metode *birthday attack* oleh D. Wagner yang dipublikasikan pada tahun 2002.

Serangan *birthday attack* adalah sebagai berikut:

Misalkan L1, …, L4 merupakan empat *list* dari n bit bilangan bulat acak. Yang harus dilakukan adalah menemukan x_i € L_i sehingga persamaan berikut terpenuhi:

$$x1 \oplus x2 \oplus x3 \oplus x4 = 0$$

Solusi dapat ditemukan dengan probabilitas yang baik jika setiap *list* mengandung setidaknya $2^{n/4}$ bilangan bulat. Langkah paling jelas yang ada adalah menggenerasi semua nilai yang mungkin dari $x1 \oplus x2 \oplus x3 \oplus x4$ dan melihat apakah terjadi kolisi. Untuk melakukan ini diperlukan kompleksitas waktu $O(2^{n/2})$.

Birthday attack dari Wagner menyelesaikan persoalan ini dalam kompleksitas waktu $O(2^{n/3})$ untuk *list* dengan ukuran setidaknya $2^{n/3}$. Pertama buat *list* dari kira-kira sebanyak $2^{n/3}$ nilai $y=x1 \oplus x2$ sedemikian sehingga n/3 *low-order* dari bit y adalah nol. Kompleksitas yang dibutuhkan adalah sebesar $O(2^{n/3})$. Hal yang sama dilakukan untuk nilai $z = x3 \oplus x4$. Didapatkan dua *list* dari $2^{n/3}$ bilangan bulat dengan n/3 *low-order* bit diset ke nilai nol. Kemudian dilihat apakah ada kolisi yang terjadi di antara kedua *list*, dan solusi ditemukan pada $O(2^{n/3})$.

Teknik ini dapat dilakukan untuk menemukan jumlah nol dari 2^a *list*, dan membutuhkan kompleksitas waktu $O(2^a.2^{n/(a+1)})$ dengan ukuran *list* $O(2^{n/(a+1)})$.

Sementara itu terdapat pula varian dari *birthday attack* yang diajukan oleh Jean-Sebastien Coron dan Antoine Joux. Serangan mereka bertujuan sama, yaitu menciptakan kolisi dari nilai *hash.* Atau didefinisikan juga menciptakan dua pesan berbeda, $m \neq m'$ sedemikian dengan nilai *hash* $H(m) = H(m')$. Maka untuk setiap w matriks dengan u kolom, maka dipilih dua kolom sehingga hasil SOR dari 2w kolom mengembalikan nilai 0.

Untuk setiap sub matriks H_i, dapat dibuat sebuah *list* Li_i dengan ukuran $u^2/2$ nilai x_i yang merupakan nilai XOR dari dua kolom pada H_i. Selanjutnya menggunakan algoritma Wagner untuk menemukan *birthday attack* pada w *list*:

$$x1 \oplus x2 \oplus \ldots \oplus x_w = 0$$

Lebih akurat lagi, untuk t sedemikian sehingga $2^t = u^2/2$. Maka ada 2^{2t} elemen $x1 \oplus x2$, di mana $x_1 \in L_1$ dan $x_2 \in L_2$, dengan 2^t sedemikian sehingga *rightmost* t bit adalah 0. Ini memberikan *list* L_1', yang bisa dibuat dalam kompleksitas $O(2^t)$. Untuk *list* (L_3, L_4) dapat dilakukan hal yang sama dan mendapatkan L' untuk keduanya.

Lalu, dengan *birthday paradox* dapat ditemukan elemen pada $L_1' \oplus L_2'$ yang memenuhi syarat 3t *rightmost* bitnya adalah nol, dengan kompleksitas waktu $O(2^t)$. Maka, jika w = 4 dan ukuran *hash* adalah r = 3t, dapat ditemukan kolisi dengan kompleksitas waktu O(2`). Nilai w dapat dibuat lebih tinggi lagi dengan membangun pohon yang saling berhubungan dan dapat ditemukan kolisi dengan kompleksitas waktu $O(w.2^t)$ jika:

$$r \leq (\log_2(w) + 1).t$$

di mana $t = 2 \log_2(u) - 1$

Namun saying sekali, hal ini tidak cukup untuk memecahkan fungsi *hash* untuk parameter yang direkomendasikan, sehingga sebelumnya harus dilakukan dahulu pengambilan seluruh 2^{2t} elemen $x1 \oplus x2$, dan menggunakan pohon dengan kedalaman yang sama dikurangi satu. Mudah untuk dilihat bahwa kolisi dapat ditemukan dengan kompleksitas waktu $O(w.2^{2t})$ jika:

$$r \leq 2(\log_2 w).t$$

Ini memecahkan instan pertama dengan r = 160, w = 64, u = 256 dan t = 15, dalam waktu 2^{36}.

Untuk instan kedua (r = 224, w = 96, u = 256, t = 15), *list* L_i dapat dikelompokkan dahulu menjadi tiga, yang memberikan 32 buah *list* berisi 245 elemen, di mana hanya diambil sejumlah 238. Bila w = 6, 2 . 38 = 76 bit dapat di-nol-kan, bila w = 12, 3 . 38 = 114 bit dapat di-nol-kan, dan dengan w = 96, 6 . 38 = 228 bit dapat di-nol-kan, yang akan memecahkan fungsi *hash* dalam waktu 32 . 238 = 243.

Untuk instan ketiga (r = 288, w = 128, u = 64, t = 11), *list* L_i dapat dikelompokkan menjadi enam, dan mengambil 2^{58} elemen dari 2^{66}. di mana hanya diambil sejumlah 2^{38}. Bila w = 12, 2 . 58 = 116 bit dapat di-nol-kan, dan dengan w = 96 < 128, 5 . 58 = 290 dapat di-nol-kan, yang akan memecahkan fungsi *hash* dalam waktu 16 . 258 = 262.

4. Analisis Kriptanalisis Fungsi Hash

Dari beberapa teknik kriptanalisis yang dibahas dapat dilihat bahwa teknik pencarian kolisi merupakan metode utama dalam penyerangan fungsi *hash.* Hal ini dikarenakan mencari *second pre-images* sangatlah memungkinkan untuk dilakukan dibandingkan dengan menginversi pesan dari nilai *hash* yang ada. Metode pencarian kolisi dari nilai *hash* pada umumnya menggunakan manipulasi operator bit yang hampir serupa dengan algoritma fungsi *hash*

yang diserang. Untuk pencarian hasilnya menggunakan percobaan dengan teknik *divide and conquere* yang dilakukan dengan mencari penghasil nilai tertentu dari hsil operasi bit dan menghilangkan variabel yang tidak mungkin. Di sini pemanfaatan peluang kemunculan hasil yang akan diselidiki variabel pembentuknya sangat penting.

Beberapa analisis yang dapat dilakukan berhubungan dengan fungsi *hash* dan kriptanalisisnya adalah sebagai berikut:

1. Untuk pembuatan fungsi *hash* yang pada umumnya mengolah *string* bit, maka perbandingan beberapa operator bit yang ada adalah sebagai berikut:
    a. Fungsi OR

    Peluang untuk mencari sejumlah n variabel pembentuk hasil dari operasi bit OR adalah $1/(2^n - 1)$, untuk nilai 1 dan 1 untuk nilai 0.

    Misal untuk hasil 1 dari operasi OR pada variabel x1, x2, x3 (3 variabel) maka peluang untuk mendapatkan bahwa hasil dari variabel (x1,x2,x3) = (0,1,0) adalah 1 dari $2^3 - 1 = 7$ kemungkinan yang ada dan dapat dijabarkan sebagai berikut:

    (0,0,1)

    (1,0,0)

    (0,1,1)

    (1,1,0)

    (1,0,1)

    (1,1,1)

    dan (0,1,0) itu sendiri.

    Sedangkan untuk mendapatkan nilai 0 satu-satunya yang dapat memenuhi adalah seluruh variabel bernilai 0 juga. Maka peluangnya adalah 1 atau mutlak.

    b. Fungsi AND

    Kebalikan dari fungsi OR, peluang untuk mencari sejumlah n variabel pembentuk hasil dari operasi bit OR adalah $1/(2^n - 1)$, untuk nilai 0 dan 1 untuk nilai 1.

    c. Fungsi XOR

    Untuk seluruh nilai, peluang untuk mencari kombinasi dari nilai sejumlah n variabel adalah sebesar $1/(2^{n-1})$. Peluang yang sama besar untuk kedua nilai bit, 1 dan 0.

    Misal untuk hasil 0 dari operasi XOR pada variabel x1, x2, x3 (3 variabel) maka peluang untuk mendapatkan bahwa hasil dari variabel (x1,x2,x3) = (1,1,0) adalah 0 dari $2^{3-1} = 4$ kemungkinan yang ada dan dapat dijabarkan sebagai berikut:

    (0,0,0)

    (1,1,0)

    (1,0,1)

    (0,1,1)

    Sehingga dari analisis tadi penggunaan operator OR dan AND lebih riskan untuk mendapat serangan kolisi. Karena dapat menghasilkan probabilitas satu yang akah memudahkan pencarian *second pre-images*. Sedangkan operator XOR selalu konstan menghasilkan probabilitas 50:50 untuk setiap nilai 1 dan 0.

2. Untuk meningkatkan kekuatan sebuah algoritma fungsi *hash,* maka sebaiknya nilai *hash* tidak dalam bentuk bit yang merupakan hasil akhir dari operasi pada algoritma. Hal ini dapat menyebabkan penyerangan lebih mudah dilakukan. Dapat dikatakan nilai *hash* seperti ini adalah tanda tangan yang polos dan mudah untuk ditelusuri variabel – variabel pembentuknya. Metode yang dapat digunakan adalah dengan

membentuk bit menjadi karakter dengan cara mensubtitusi ataupun transposisi. Hal ini bisa menjadi pertahanan tambahan dan menambah kesulitan penyerangan fungsi *hash.*

Penggunaan teknik putaran secara iteratif juga mempengaruhi tingkat kerumitan nilai *hash*, namun terkadang juga malah mempermudah kemungkinan untuk menelusuri algoritma secara terbalik.

Namun perlu diperhatikan pula tingkat kecepatan pembangunannya. Karena tentu saja kita tidak menginginkan hasil nilai *hash* yang sangat sulit dimengerti namun pembuatannya sangat lambat.

3. Bila dipikirkan secara logika, semangkus apapun fungsi *hash* yang dibuat maka pastilah sedemikian rupa dapat menimbulkan kolisi antar nilai *hash*. Pembuktian untuk hal ini dapat dilakukan secara matematis.

   Fungsi *hash* umumnya membuat nilai *hash* yang ukurannya jauh lebih kecil dari pesan. Misal panjang bit nilai *hash* adalah h dan panjang bit pesan adalah n. Maka peluang untuk membangun nilai *hash* adalah 2^h. Sedangkan kemungkinan pesan adalah 2^n. Untuk nilai h yang lebih kecil dari n maka tidaklah mungkin untuk membuat pemetaan satu-satu dari nilai *hash* ke pesan dan juga sebaliknya. Karena nilai h yang lebih kecil, maka mungkin ada lebih dari satu pesan yang dipetakan ke nilai *hash* yang sama. Secara matematis kemungkinan ada $2^n - 2^h$ pesan yang memiliki nilai *hash* yang berkolisi dengan 2^h pesan lainnya.

   Tapi tentunya tidak semua pesan sejumlah $2^n - 2^h$ tersebut yang memiliki arti. Tentu dapat dipastikan sebuah pesan memiliki tanda tangan palsu bila isi pesan tersebut sangat kacau dan tidak memiliki arti.

   Bila dilakukan perhitungan peluang bahwa sebuah pesan yang berkolisi memiliki arti atau tidak (dalam kasus untuk pesan bahasa Indonesia), maka misalkan dari elemen paling kecil dari pesan, yaitu kata. Untuk kata yang terdiri dari 3 huruf kemungkinan pembentukannya adalah 26^3. Namun agar memiliki arti minimal harus mengandung satu huruf vokal namun tidak tiga huruf vokal. Maka kata yang mungkin memilii arti adalah sebesar:

   $26^3 - 21^3 - 5^3 = 8.190$ kemungkinan

   Namun tentu saja tidak seluruhnya memiliki arti yang benar. Misalnya kita anggap seperempat dari jumlah tersebut, yaitu $\pm$ 2000. Maka untuk mendapatkan sebuah pesan yang kemungkinan besar memiliki arti adalah 2000: 26^3 atau $\approx$ 0.11.

   Karena itu masih sangat mungkin untuk menemukan irisan antara pesan bentukan yang memiliki arti dengan probabilitas 0.11 dan pesan yang berkolisi sebesar $2^n - 2^h$.

   Fungsi *hash* bukanlah fungsi yang dapat berdiri sendiri. Bila ada aplikasi tanda tangan digital yang hanya menggunakan fungsi *hash* sebagai pencipta tanda tangannya, maka dapat dipastikan bahwa aplikasi tersebut tidak aman. Bahkan bila menggunakan fungsi *hash* terbaru sekali pun. Fungsi *hash* memerlukan integrasi dengan fungsi penunjang lain seperti misalnya kunci publik dan privat. Atau lebih berguna sebagai fungsi penunjang seperti dalam penggenerasian kunci untuk menyamakan panjang kunci yang digunakan.

5. Kesimpulan

Untuk aplikasi yang mengandung fungsi *hash* harus diperhatikan apakah serangan kolisi acak dapat dilakukan terhadap aplikasi tersebut. Bila hal tersebut dapat dilakukan, analisis resiko yang tepat harus dilakukan. Bila efek yang diakibatkan sebuah serangan yang sukses bukanlah sebuah faktor serius yang berpengaruh, maka fungsi *hash* atau pun aplikasi tersebut tidak masalah untuk digunakan. Namun bila sebaliknya, maka penggantian fungsi *hash* tentu saja harus dilakukan. Untuk aplikasi yang lebih baru sebaiknya fungsi *hash* tersebut tidak digunakan.

Untuk saat ini batasan pilihan lebih banyak untuk menggunakan ekstensi atau varian dari SHA-1, tidak termasuk SHA-1 itu sendiri. Ini berbeda dari situasi sebelum ditemukannya serangan karena SHA-1, yang sebelumnya direkomendasikan digunakan hingga tahun 2010 tidak akan direkomendasikan lebih lama lagi. NIST merekomendasikan untuk menggunakan SHA-256, dan tidak lagi menyarankan penggunaan SHA-1 hingga 2010.

Namun tampaknya ini bukanlah akhir dari cerita perkembangan fungsi *hash.* Apakah sekarang kita dapat dengan tenang menggunakan SHA-256 dan berharap bahwa algoritma ini adalah yang terbaik? Atau apakah seharusnya kita memperbaiki seluruh kekurangan yang ada dan merombak total metode *hash* yang digunakan dan kembali dengan metode yang lebih baik? Mari coba perhatikan dahulu situasi yang ada secara lebih detail. Seluruh fungsi *hash* yang standar digunakan seperti MD4, MD5, dan SHA-0/1/224/256/384/512, mengikuti cara dasar yang sama, dengan hanya sedikit perubahan pengaturan dan konfigurasi bagian SHA-1 dari MD4, MD5, dan SHA-0, dan dengan tambahan pengacakan rumit pada SHA-224/256/384/512.

Di penghujung tahun 2004, sepertinya ada harapan bahwa dapat dilakukan perubahan kecil pada sistem proteksi fungsi *hash* untuk melawan serangan kolisi yang diumumkan pada tahun 2004. Namun setelah pada bulan Februari 2005 diumumkan kriptanalisis terhadap SHA-1, maka harapan tersebut pupus sudah. Dengan pendahulunya yang telah terbukti memiliki desain iteratif yang kurang sehat, maka kita tinggal menunggu hingga berapa lama lagi SHA-224/256/384/512 dapat bertahan?

Dari "*Further progress in hashing cryptanalysis"*, Adi Shamir, salah satu dari kriptanalisis yang paling ditakuti dan mendapat banyak respek dari para kriptografer, merekomendasikan memulai sebuah sketsa baru dalam duni *hashing.* Namun dalam waktu yang dekat ini tampaknya SHA-256/384/512 akam menjadi pilihan fungsi *hash.* Dengan pengembangan saat ini, protocol desain perlu dikembangkan untuk perubahan fungsi *hash,* seperti yang telah banyak dilakukan, dan ketika telah dilakukan maka perlu dipastikan bahwa serangan versi kebalikan dari algoritma dapat dicegah pula.

Tampaknya memang perlu dirancang ulang kembali fungsi *hash* yang ada dengan keamanan yang baru. Sesuai dengan pendapat para ahli teknologi bahwa teknologi selalu berkembang berdasarkan kekurangannya sendiri, dan kedua hal ini selalu adu cepat. Maka dari analisis sebelumnya dapat disimpulkan beberapa hal berikut:

1. Untuk operasi bit operator XOR sangat efektif untuk digunakan.
2. Pembuatan putaran iteratif perlu ditinjau apakah menguntungkan dengan kerumitan yang diciptakan atau malah mengurangi keamanan serangan kolisi karena terlalu terintegrasi dan lebih cepat dipecahkan.
3. Fungsi *hash* yang telah terpecahkan masih dapat digunakan sebagai fungsi penunjang seperti penggenerasi kunci dan penyetaraan ukuran pesan.

Referensi

www.mail.informatika.org/~rinaldi.

http://www.cs.ucsd.edu/users/bsy/dobbertin.ps.

http://www.cosic.esat.kuleuven.be/publications /article-52.ps.

http://eprint.iacr.org/2005/010.

http://www.scheiner.com/blog/archives/2005/02/ sha_1broken.html.

Coron, Jean-Sebastian dan Antoine Joux, Cryptanalysis of a Provably Secure Cryptographic Hash Function, Gemplus Card International.

Lenstra, Arjen K., Further progress in hashing cryptanalysis, Lucent Technologies, Bell Laboratories, 2005.