Analisis dan Perancangan Modifikasi dari Algoritma MD5 untuk Menghindari Terjadinya Kolisi

Giovanni Sakti Nugraha [1)]
1) Program Studi Informatika ITB, Bandung 40132, email: if14096@students.if.itb.ac.id

Abstract – *MD5 merupakan salah satu algoritma hashing yang paling populer digunakan. Pada umumnya, MD5 digunakan sebagai digital signature atau tanda tangan digital dari sebuah dokumen. Tangan tangan digital digunakan untuk melakukan verfikasi terhadap dokumen tersebut dan dapat dimanfaatkan untuk melakukan pemeriksaan apakah integritas dokumen tersebut terjaga dan dikirim oleh orang yang bersangkutan atau tidak. Namun pada tahun 2004, Xiaouyun Wang dan Hongbu Yu berhasil menemukan algoritma yang cukup mangkus untuk menemukan kolisi dari algoritma MD5. Kedua orang kriptografer tersebut menemukan algoritma yang mampu menemukan 2 dokumen berbeda namun memiliki nilai hash MD5 yang sama. Hal ini tentunya sangat berbahaya karena orang yang tidak bertanggung jawab akan mampu memanfaatkannya untuk membentuk dokumen palsu yang disamarkan sebagai dokumen asli untuk memperdaya korbannya. Untuk itu perlu dilakukan upaya modifikasi terhadap algoritma MD5 agar terjaga tingkat keamanannya dengan mencegah kemungkinan terjadinya kolisi. Makalah ini akan membahas mengenai algoritma MD5 secara umum, kelemahan dari MD5 yang telah berhasil dieksploitasi, kemungkinan serangan brute force dan birthday attack terhadap MD5 dan solusi yang ditawarkan untuk meningkatkan keamanan MD5.*

Kata Kunci*: hash, digital signature, MD5, MD5 collision, birthday attack*

1. PENDAHULUAN

Dokumen tertulis dalam dunia teknologi informasi yang semakin berkembang pesat merupakan entitas yang perlu dijaga kerahasiaannya hanya bagi orang-orang yang berkepentingan. Selain itu dokumen juga harus dapat dijaga keasliannya agar tidak ada orang yang tidak berkepentingan merubah dokumen. Oleh karena itu selain algoritma untuk melakukan enkripsi, juga terdapat algoritma untuk melakukan *hashing* yang salah satu fungsinya adalah untuk menjaga keaslian dokumen tersebut. Algoritma MD5 merupakan salah satu algoritma *hashing* yang diciptakan untuk keperluan tersebut. Namun hingga saat ini telah ditemukan beberapa kelemahan untuk melakukan serangan terhadap algoritma MD5. Serangan kepada algoritma MD5 dalam hal ini berarti serangan untuk menciptakan kolisi, yaitu keadaan dimana terdapat dua buah dokumen yang berbeda namun memiliki nilai *hash* yang sama. Hal ini dapat dimanfaatkan oleh orang yang tidak berkepentingan untuk membuat dokumen palsu dengan mengatasnamakan dokumen yang asli.

Bagian pertama pada makalah ini akan menjelaskan mengenai algoritma MD5 secara lebih terperinci., bagian kedua akan menjelaskan mengenai serangan-serangan terhadap kelemahan algoritma MD5 yang telah ditemukan, bagian ketiga akan menjelaskan mengenai modifikasi yang dilakukan untuk menghindari kolisi serta bagian selanjutnya akan menjelaskan mengenai eksperimen yang dilakukan beserta perbandingannya dengan algoritma MD5 konvensional.

2. PENJELASAN ALGORITMA MD5

2.1. Deskripsi Umum

MD5 merupakan fungsi *hash* yang diciptakan oleh Ronald Rivest pada tahun 1991. MD5 merupakan suksesor dari algoritma MD4 yang telah berhasil diserang oleh kriptanalisis. Ciri dari algoritma MD5 adalah menghasilkan pesan berupa *message digest* yang panjangnya 128 bit dari pesan orisinal yang menjadi masukannya.

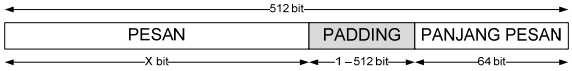
2.2. Langkah-langkah Pembuatan *message digest* dengan algoritma MD5

Terdapat 4 langkah utama yang merupakan proses pembuatan *message digest* oleh algoritma MD5, yaitu mencakup penambahan bit-bit pengganjal (padding) terhadap pesan, penambahan nilai panjang pesan semula, inisialisasi penyangga MD5 dan pengolahan pesan dalam blok berukuran 512 bit.

2.2.1. Penambahan bit-bit pengganjal

Sebelum dilakukan pemrosesan lebih lanjut, pesan yang menjadi masukan, terlebih dahulu harus ditambah dengan bit-bit pengganjal agar panjang pesan menjadi kelipatan 512 bit kurang 64 bit. Hal ini dilakukan karena MD5 memproses pesan dalam blok-blok berkelipatan 512 bit, dan 64 bit sisanya yang tidak dijadikan bagian dari pesan merupakan tempat

yang disediakan untuk menuliskan panjang pesan. Sebuah kasus unik yang perlu diperhatikan adalah jika panjang pesan pada blok terakhir tepat 480 bit, maka perlu ditambahkan 512 bit pengganjal untuk kasus tersebut.



Gambar 1-1 Skema sebuah blok pada MD5

Pengganjal yang diberikan pada pesan tersebut berupa kumpulan bit 1 dan bit 0 dengan urutan bit 1 diletakkan di awal dan bit 0 diletakkan hingga pesan mencapai panjang dengan kelipatan 512 bit kurang 64 bit.

2.2.2. Penambahan nilai panjang pesan semula

Setelah pesan diberikan bit pengganjal, maka panjang pesan akan menjadi kelipatan 512 bit kurang 64 bit. Bagian akhir yang panjangnya 64 bit tersebut kemudian akan diisi dengan panjang pesan yang menjadi masukkan (pesan awal yang belum ditambahkan bit pengganjal). Jika terjadi kasus dimana panjang pesan lebih panjang daripada 2^{64} maka panjang pesan yang dituliskan adalah panjang pesan yang telah dimodulo dengan 2^{64}.

2.2.3. Inisialisasi penyangga MD5

MD5 membutuhkan penyangga (*buffer*) berupa sekumpulan kombinasi karakter yang menyerupai *initialization vector* (IV) seperti pada algoritma kunci simetri. Nilai-nilai penyangga tersebut akan diproses bersama-sama dengan pesan yang menjadi masukan.

Nilai-nilai penyangga algoritma MD5 yang menjadi standar sesuai dengan spesifikasi yang diberikan oleh Ronald Rivest adalah sebagai berikut :

```
A = 67452301
B = EFCDAB89
C = 98BADCFE
D = 10325467
```

Beberapa varian dari algoritma MD5 menggunakan jenis penyangga yang berbeda, namun untuk makalah ini akan digunakan nilai penyangga yang telah disebutkan di atas agar sesuai dengan spesifikasi algoritma MD5 yang standar.

2.2.4. Pengolahan pesan dalam blok berukuran 512 bit

Setelah pesan diproses menjadi pesan dengan panjang yang berkelipatan 512 bit, proses berikutnya adalah membagi pesan-pesan tersebut menjadi beberapa blok yang berukuran 512 bit. Masing-masing blok tersebut kemudian diproses dengan penyangga sehingga menghasilkan keluaran 128 bit.

Tiap proses yang dilakukan memiliki 4 buah putaran dan tiap putaran melakukan operasi yang telah dispesifikasikan sebanyak 16 kali dengan memanfaatkan elemen dari sebuah tabel nilai radian dengan rumus $t(i) = 2^{32} \times abs(sin(i))$. Untuk tiap operasi yang selesai, penyangga yang telah diproses tersebut kemudian digeser ke kanan secara sirkuler. Nilai akhir dari algoritma MD5 merupakan konkatenasi dari keempat penyangga yang telah diproses tersebut.

2.3. Aplikasi kriptografi yang memanfaatkan algoritma MD5

Aplikasi-aplikasi yang dijelaskan dalam hal ini merupakan aplikasi yang umum digunakan dan memanfaatkan algoritma MD5 beserta kemungkinan serangan yang dapat dilakukan untuk masing-masing kasus.

2.3.1 Protokol tanda tangan digital tanpa enkripsi dan dengan enkripsi

Algoritma MD5 dapat digunakan secara langsung untuk mendapatkan nilai *message digest* dari dokumen yang bersangkutan. Nilai *message digest* tersebut kemudian dapat dikirimkan bersama-sama dokumen sehingga penerima dokumen dapat melakukan verifikasi terhadap keaslian dokumen[2]. Serangan pada aplikasi jenis ini dapat dilakukan dengan membuat kolisi dari dokumen tersebut dan mengirimkan dokumen yang palsu dengan mengatasnamakan dokumen yang asli. Teknik pembentukan kolisi dapat dilihat pada bab selanjutnya.

Untuk meningkatkan keamanan, *message digest* dapat dienkripsi terlebih dahulu sebelum dikirimkan bersama-sama dokumen[2]. Hal ini tentunya meningkatkan keamanan dan orang yang tidak bertanggung jawab harus menemukan terlebih dahulu kunci publik dari *message digest* yang telah dienkripsi. Serangan harus dilakukan dengan dua tahap, yaitu menemukan kunci publik dari algoritma enkripsi (seperti RSA) dan menemukan kolisi dari *message digest* yang ditemukan. Hal yang mungkin menjadi kekurangan dari metode ini adalah mengurangi kepraktisan karena harus ada pembagian kunci publik terlebih dahulu hanya kepada orang yang berhak beserta proses enkripsi yang cukup memakan waktu jika terdapat banyak *message digest* yang harus diamankan.

2.3.2. Penyimpanan *password* untuk aplikasi *multiuser*

Aplikasi *multiuser* membutuhkan penyimpanan untuk data-data pribadi pengguna yang mengaksesnya. Data-data yang ingin disimpan tersebut mencakup kata kunci atau *password* yang berguna bagi si pengguna untuk melakukan *login*. Tentunya sangat berbahaya

jika *password* tersebut disimpan sebagai *plaintext* biasa di dalam *database*. Enkripsi terhadap *password* juga dapat dilakukan namun keamanan tetap tidak dapat dijamin, karena *admin* dari aplikasi akan dapat mengetahui *password* dari masing-masing user dengan cara seperti itu. Cara yang paling aman adalah dengan memanfaatkan fungsi *hash* satu arah seperti MD5. Proses dilakukan dengan mendapatkan nilai *hash* dari *password*, menyimpannya ke dalam *database*. Jika ada pengguna yang ingin melakukan *login*, nilai *hash* diambil dari masukan, kemudian dicocokan dengan nilai *hash* yang ada di dalam *database* untuk *user* tersebut.

Serangan pada aplikasi ini dapat dilakukan dengan percobaan untuk mendapatkan *plaintext* dari nilai *hash* yang ditemukan, sehingga penyerang dapat *login* sebagai pengguna yang bersesuaian.

2.3.3. Verifikasi keabsahan sebuah file

Dalam pendistribusian *file-file* dan arsip di internet, diperlukan sebuah protokol untuk menjaga orisinalitas suatu *file*. Karena bisa saja terdapat virus atau *malware* dan *spyware* yang mengatasnamakan sebagai suatu *file* yang banyak diunduh oleh pengguna internet. Oleh karena itu, dalam pendistribusian, umumnya *file* disertakan dengan sebuah nilai *hash* dan pengguna dapat melakukan verifikasi terhadap *file* terlebih dahulu sebelum menjalankannya. Serangan dapat dilakukan dengan membuat kolisi dari *executable* sehingga pengguna akan menganggap *file* yang ia jalankan valid walaupun sesungguhnya tidak.

3. SERANGAN YANG DAPAT DILAKUKAN TERHADAP ALGORITMA MD5

Serangan yang dimaksud pada bagian ini adalah serangan untuk membentuk kolisi dari algoritma MD5 dan serangan untuk mengetahui plaintext yang bersesuaian dari nilai *hash* yang dihasilkan.

3.1. *brute force attack*

Serangan *brute force* merupakan serangan yang dilakukan dengan mencoba seluruh kemungkinan dari ruang pencarian. Serangan *brute force* terhadap algoritma MD5 umumnya ditujukan untuk mengetahui plainteks dari *password* yang disimpan di dalam tempat penyimpanan seperti *database* sebagai kode *hash* MD5 oleh aplikasi-aplikasi web maupun *server*. Untuk MD5, serangan ini dapat dilakukan dengan menebak (*guessing*) atau mencoba dengan bantuan *dictionary* khusus, berisi kata-kata yang umum digunakan sebagai *password*.

Serangan juga dapat dilakukan dengan bantuan *rainbow table*, sebuah teknik pertukaran waktu komputasi dengan tempat penyimpanan yang mampu melakukan serangan *brute force* dengan lebih cepat karena menyimpan kumpulan *pre-computed* data pada sebuah tabel yang dapat mempercepat proses pencarian kunci yang sesuai.

Kedua serangan di atas dapat diatasi dengan menggunakan *salt*, yaitu string atau bit tambahan yang diberikan di akhir plainteks. Namun penggunaan *salt* juga tetap memiliki resiko karena panjang nilai *hash* algoritma MD5 yang terlampau pendek untuk kekuatan komputasi yang dimiliki manusia pada saat ini. Hal ini didukung fakta bahwa terkadang *salt* yang digunakan oleh aplikasi yang bersangkutan merupakan hal yang berkaitan dengan *username* pemilik *password* sehingga *salt* tersebut mudah untuk diterka.

3.2. *birthday attack*

Serangan ulang tahun atau yang umum dikenal sebagai *birthday attack* merupakan serangan yang mirip dengan *brute force* namun memiliki tujuan yang lebih spesifik, yaitu menemukan 2 buah *plaintext* yang memiliki nilai *hash* serupa atau yang disebut sebagai kolisi sebagaimana yang telah dijelaskan di atas[3].

Serangan dilakukan dengan menemukan x1 dan x2 dimana fungsi x akan memetakan x1 dan x2 dan mendapatkan hasil yang serupa. Sesuai dengan *birthday paradox*, hal ini sebenarnya memiliki kemungkinan yang cukup besar dan dapat dihitung secara *brute force*, terlebih lagi algoritma MD5 memiliki keluaran yang panjangnya lebih sedikit jika dibandingkan dengan algoritma fungsi *hash* lainnya seperti SHA1[4].

3.4. Serangan Lenstra,Wang dan deWeger

Ketiga peneliti dari masing-masing universitas yang berbeda, yaitu Lenstra, Wang dan de Weger berhasil memanfaatkan kelemahan algoritma MD5 yang berhasil mereka temukan untuk menciptakan sebuah kolisi pada tahun 2004. Mereka mempublikasikan temuannya dengan memberikan dua buah dokumen sertifikasi X.509 yang berbeda namun memiliki *message digest* serupa[1]. Kedua dokumen tersebut diciptakan dengan algoritma yang mereka bentuk memanfaatkan sebuah superkomputer IBM P960 dalam waktu satu jam[6]. Algoritma yang digunakan tidak pernah dipublikasikan, namun terdapat *statement* yang menyatakan bahwa kolisi pada MD5 terjadi karena lemahnya *avalanche effect* yang dimiliki oleh fungsi-fungsi MD5[5]. Meskipun belum masuk tahap *practical* karena membutuhkan sebuah superkomputer, penelitian ini kelak akan menjadi dasar untuk percobaan pembentukan kolisi berikutnya.

3.5. Serangan Klinina

Vlastimil Klinina mempublikasikan serangannya dalam pembentukan kolisi dari MD5 dengan hanya

memanfaatkan sebuah komputer laptop saja. Klinina menyempurnakan algoritma pencarian kolisi yang dimiliki oleh Wang dkk. dan menjalankannya dalam tahap *practical* yaitu hanya memanfaatkan sebuah komputer personal yang umum dimiliki oleh perseorangan.

4. MODIFIKASI DAN IMPLEMENTASI DARI MODIFIKASI YANG DILAKUKAN PADA ALGORITMA MD5

4.1. Memperpanjang nilai *hash* yang dihasilkan

Memperpanjang nilai *hash* yang dihasilkan berguna untuk mempersulit serangan yang memanfaatkan metode *brute force* atau *birthday attack*. Dengan demikian dibutuhkan kemampuan komputasi secara eksponensial yang lebih besar untuk mengetahui *plaintext* maupun kolisi dari sebuah nilai *hash*.

Modifikasi algoritma ini akan meningkatkan keluaran yang dihasilkan oleh algoritma MD5 menjadi 4 kali lipat, yaitu sebesar 512 bit.

4.1.1. Modifikasi penambahan bit-bit pengganjal

Untuk memperpanjang nilai *hash*, dibutuhkan bit-bit pengganjal yang lebih banyak. Untuk itu, penambahan bit pengganjal tetap dilakukan dengan bit 1 dan 0, namun dilakukan hingga panjang pesan mencapai kelipatan 2048 bit kurang 64 bit.

4.1.2. Merubah nilai penyangga yang digunakan

Agar keluaran yang diharapkan panjangnya menjadi 512 bit, maka penyangga yang diberikan juga harus dimodifikasi agar sesuai. Berikut adalah nilai penyangga standar yang baru.

```
A = 67452301 76543210
B = EFCDAB89 FEDCBA98
C = 98BADCFE 89ABCDEF
D = 10325467 01234567
E = 76543210 67452301
F = FEDCBA98 EFCDAB89
G = 89ABCDEF 98BADCFE
H = 01234567 10325467
```

4.1.3. Merubah tabel nilai radian

Tabel nilai radian yang digunakan juga perlu dimodifikasi untuk menyesuaikan dengan panjang nilai penyangga yang baru. Rumus untuk tabel nilai radian yang baru adalah $t(i) = 2^{64} \times abs(sin(i))$.

4.2. Melakukan lebih dari satu kali *pass* terhadap nilai *hash*

Untuk meningkatkan *avalanche effect* dari algoritma MD5, akan dilakukan lebih dari satu kali *pass* terhadap nilai *hash* yang dihasilkan. Setiap selesai sebuah *pass*, nilai *hash* tidak langsung dikonkatenasi, melainkan akan diproses lagi dan nilai-nilai *hash* tersebut menjadi nilai penyangga yang baru. Proses tersebut akan dilakukan sebanyak dua iterasi, sehingga terdapat tiga iterasi utama untuk algoritma baru ini.

5. EKSPERIMEN DAN PERBANDINGAN ALGORITMA MD5 YANG TELAH DIMODIFIKASI DENGAN ALGORITMA MD5 KONVENSIONAL

5.1. Analisis terhadap modifikasi algoritma MD5 yang dihasilkan

Menurut data yang dimiliki, untuk melakukan serangan terhadap algoritma dengan keluaran 512 bit ini baik secara *brute force* maupun *birthday attack* membutuhkan kalkulasi *hashing* sebanyak 1.4×10^{77} untuk mendapatkan kemungkinan berhasil sebesar 50% dari kalkulasi yang dilakukan [3]. Jika menggunakan sebuah superkomputer dengan kemampuan kalkulasi 10^{16} per detik pun membutuhkan waktu 1.62×10^{56} tahun yang tentunya cara ini tidak dapat diterapkan.

Algoritma Wang, dkk. dan Klinina memang tidak dapat diujicobakan karena algoritma yang mereka miliki belum pernah dipublikasikan. Namun menurut kelemahan yang dikemukakan, yaitu lemahnya *avalanche effect* dari algoritma MD5 konvensional, masalah ini dicoba untuk diatasi oleh algoritma baru ini dengan melakukan lebih dari sebuah *pass* terhadap perputaran utama dari algoritma. Diharapkan dengan proses tersebut dapat meningkatkan kekuatan *avalanche effect* dari algoritma dan mencegah pembentukan kolisi.

5.2. Eksperimen yang dilakukan

Langkah-langkah yang dilakukan dalam eksperimen adalah melakukan ujicoba terlebih dahulu terhadap algoritma MD5 konvensional dengan pembuatan dokumen kolisi menggunakan bantuan sebuah perangkat lunak yang bernama MD5CRK. MD5CRK merupakan aplikasi yang mampu menciptakan dua buah file eksekusi yang berbeda namun memiliki *message digest* yang sama. Terbukti bahwa algoritma MD5 konvensional mampu mendapatkan dua buah file eksekusi kolisi dengan waktu yang cukup singkat menggunakan komputer personal biasa.

Setelah itu rancangan implementasi algoritma MD5 yang telah dimodifikasi diterapkan pada aplikasi berbahasa C# dengan bantuan .NET. Aplikasi ini sukses membangkitkan *message digest* sepanjang 512 bit untuk seluruh *test case* yang diberikan. Tentunya panjang *message digest* ini memiliki ruang pencarian yang lebih besar untuk penerapan metode *brute force* dan *birthday attack* dalam melakukan serangan. MD5CRK tidak dapat diuji terhadap algoritma yang telah dimodifikasi ini, namun dengan perbedaan skema dan langkah-langkah yang dilakukan, aplikasi-aplikasi pembuatan dokumen kolisi seperti MD5CRK harus didesain ulang untuk menangani modifikasi dari algoritma MD5 ini.

5.3. Perbandingan dengan algoritma MD5 konvensional

Algoritma MD5 konvensional yang memiliki keluaran sebesar 128 bit, membutuhkan komputasi *hashing* sebanyak 2.2×10^{19} untuk mendapatkan kemungkinan berhasil sebesar 50% [3]. Jika menggunakan sebuah superkomputer dengan kemampuan kalkulasi sebanyak 10^{16} per detik maka membutuhkan waktu selama stengah jam hingga satu jam. Walaupun belum ada superkomputer yang dapat melakukan perhitungan MD5 secepat itu, namun teknologi untuk menerapkannya sudah dapat diterapkan, mengingat kemampuan komputasi dari komputer yang dibuat oleh manusia semakin meningkat pesat tiap waktunya.

MD5 konvensional juga terbukti telah dapat dibuat kolisisnya dengan bantuan algoritma Wang dan Klinina, sedangkan untuk algoritma baru ini walaupun belum diuji secara seluruhnya memiliki peluang yang lebih besar untuk menghindari permasalahan tersebut karena kelemahan yang tertulis pada beberapa literatur telah diatasi.

6. KESIMPULAN

Semakin meningkatnya kemungkinan serangan untuk algoritma MD5 konvensional menyebabkan algoritma tersebut menjadi solusi yang kurang efektif dalam menjalankan fungsi-fungsinya. Kelemahan-kelemahan seperti pendeknya panjang *message digest* yang sudah tidak sesuai dengan kemampuan komputasi saat ini serta lemahnya *avalanche effect* menyebabkan beberapa peneliti berhasil menemukan algoritma untuk menciptakan kolisi dengan cepat dan mudah menggunakan komputer biasa sekalipun. Algoritma yang dikembangkan ini mencoba untuk mengatasi permasalahan-permasalahan tersebut dengan meningkatkan keluaran *message digest* menggunakan beberapa metode serta meningkatkan jumlah *passing* yang dilakukan. Perbaikan ini diharapkan mampu meningkatkan keamanan dan memperbaiki kekurangan yang ditemui pada algoritma MD5 konvensional.

DAFTAR REFERENSI

[1] Lenstra, Arjen., Wang, Xiaoyun., de Weger, Benne., “Colliding X.509 Certificates”, Technische Universiteit Eindhoven, 2005.

[2] Lysyanskaya, Anna., “Signature Schemes and Applications to Cryptographic Protocol Design”, Massachussets Institute of Technology, 2002.

[3] Bellare, Mihir., “Hash Function Balance and its impact on Birthday Attacks”, Eurocrypt, 2004.

[4] Randall, James., Syzdlo, Michael., “Collisions for SHA0, MD5, HAVAL, MD4 and RIPEMD but SHA1 Still Secure”, RSA Laboratories, 2004.

[5] Ir. Rinaldi Munir, M.T., “Kriptografi”, Institut Teknologi Bandung, 2006.

[6] Wang, Xiaoyun., Feng, Dengguo., Lai, Xuejia., Yu, Hongbu., “Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD”, Shandong University China, 2004.

[7] Wang, Xiaoyun., Yu, Hongbu., “How to Break MD5 and Other Hash Functions”, Shandong University China, 2004.

LAMPIRAN A

Pseudocode Algoritma MD5 (RFC1321) Yang digunakan untuk pengujian

```
//Note: All variables are unsigned 32 bits and wrap modulo 2^32 when calculating
var int[64] r, k

//r specifies the per-round shift amounts
r[ 0..15] := {7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22}
r[16..31] := {5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20}
r[32..47] := {4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23}
r[48..63] := {6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21}

//Use binary integer part of the sines of integers (Radians) as constants:
for i from 0 to 63
    k[i] := floor(abs(sin(i + 1)) × (2 pow 32))

//Initialize variables:
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476

//Pre-processing:
append "1" bit to message
append "0" bits until message length in bits = 448 (mod 512)
append bit (bit, not byte) length of unpadded message as 64-bit little-endian
integer to message

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

    //Initialize hash value for this chunk:
    var int a := h0
    var int b := h1
    var int c := h2
    var int d := h3

    //Main loop:
    for i from 0 to 63
        if 0 = i = 15 then
            f := (b and c) or ((not b) and d)
            g := i
        else if 16 = i = 31
            f := (d and b) or ((not d) and c)
            g := (5×i + 1) mod 16
        else if 32 = i = 47
            f := b xor c xor d
            g := (3×i + 5) mod 16
        else if 48 = i = 63
            f := c xor (b or (not d))
            g := (7×i) mod 16

        temp := d
        d := c
        c := b
        b := b + leftrotate((a + f + k[i] + w[g]) , r[i])
        a := temp
```

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

var int digest := h0 append h1 append h2 append h3 //(expressed as little-endian)

  //leftrotate function definition
  leftrotate (x, c)
      return (x << c) or (x >> (32-c));
```

LAMPIRAN B

Pseudocode Modifikasi Algoritma MD5 Yang digunakan untuk pengujian

```
//Note: All variables are unsigned 64 bits and wrap modulo 2^64 when calculating
var int[64] r, k

//r specifies the per-round shift amounts
r[ 0..15] := {7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22}
r[16..31] := {5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20}
r[32..47] := {4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23}
r[48..63] := {6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21}

//Use binary integer part of the sines of integers (Radians) as constants:
for i from 0 to 63
    k[i] := floor(abs(sin(i + 1)) × (2 pow 64))

//Initialize variables:
var int h0 := 0x6745230176543210
var int h1 := 0xEFCDAB89FEDCBA98
var int h2 := 0x98BADCFE89ABCDEF
var int h3 := 0x1032547601234567
var int h4 := 0x7654321067452301
var int h5 := 0xFEDCBA98EFCDAB89
var int h6 := 0x89ABCDEF98BADCFE
var int h7 := 0x0123456710325476

//Pre-processing:
append "1" bit to message
append "0" bits until message length in bits = 1984 (mod 2048)
append bit (bit, not byte) length of unpadded message as 64-bit little-endian
integer to message

//Run the iteratively “3-pass” function to generate more powerful “avalanche
effect”
for j from 0 to 2

//Process the message in successive 2048-bit chunks:
for each 2048-bit chunk of message
    break chunk into sixteen 64-bit little-endian words w[i], 0 = i = 15

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

    //Main loop:
    for i from 0 to 63
        if 0 = i = 15 then
            temp_f1 := (b and c) or ((not b) and d)
            temp_g1 := i
            temp_f2 := (f and g) or ((not f) and h)
            temp_g2 := i
        else if 16 = i = 31
            temp_f1 := (d and b) or ((not d) and c)
            temp_g1 := (5×i + 1) mod 16
            temp_f2 := (h and f) or ((not h) and g)
            temp_g2 := (5×i + 1) mod 16
        else if 32 = i = 47
            temp_f1 := b xor c xor d
            temp_g1 := (3×i + 5) mod 16
            temp_f2 := f xor g xor h
```

```
            temp_g2 := (3×i + 5) mod 16
        else if 48 = i = 63
            temp_f1 := c xor (b or (not d))
            temp_g1 := (7×i) mod 16
            temp_f2 := g xor (f or (not h))
            temp_g2 := (7×i) mod 16

        temp := h
        h := g
        g := f
        f := f + leftrotate((e + temp_f2 + k[i] + w[temp_g2],r[i]))
        e := d
        d := c
        c := b
        b := b + leftrotate((a + temp_f1 + k[i] + w[temp_g1]) , r[i])
        a := temp

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

//after 3-pass function
var int digest := h0 append h1 append h2 append h3 append h4 append h5 append h6
append h7//(expressed as little-endian)

  //leftrotate function definition
  leftrotate (x, c)
      return (x << c) or (x >> (32-c));
```

LAMPIRAN C

Contoh Implementasi Modifikasi Algoritma MD5 dalam Bahasa C#

```
using System;
using System.Collections;
using System.IO;
using System.Text;
using Org.BouncyCastle.Math;
namespace MD5ext
{
    class MD5ext
    {
        //definisikan S
        uint[] s =      {7,12,17,22,   7,12,17,22,   7,12,17,22,   7,12,17,22,
                         5,9,14,20,    5,9,14,20,    5,9,14,20,    5,9,14,20,
                         4,11,16,23,   4,11,16,23,   4,11,16,23,   4,11,16,23,
                         6,10,15,21,   6,10,15,21,   6,10,15,21,   6,10,15,21};

        public void ExecuteMD5ext(String text)
        {
            //inisialisasi penyangga
            uint A = 0x6745230176543210;
            uint B = 0xefcdab89fedcba98;
            uint C = 0x98badcfe89abcdef;
            uint D = 0x1032547601234567;
                    uint E = 0x7654321067452301;
                    uint F = 0xfedcba98efcdab89;
                    uint G = 0x89abcdef98badcfe;
                    uint H = 0x0123456710325467;

            //mengubah text ke array of byte
            byte[] container = new byte[text.Length];
            for (int i = 0; i < text.Length; i++)
            {
                container[i] = (byte)text[i];
            }

            //hitung padding yang dibutuhkan
            int temp = (1984 - ((container.Length * 8) % 2048)); //temporary
            uint pad = (uint)((temp + 2048) % 2048);

            if (pad == 0)
                pad = 2048;

            //buat message buffer
            uint sizeMsgBuff = (uint)((container.Length) + (pad / 8) + 8);
            ulong sizeMsg = (ulong)container.Length * 8;

            byte[] bMsg = new byte[sizeMsgBuff];

            for (int i = 0; i < container.Length; i++)
                bMsg[i] = container[i];

            bMsg[container.Length] |= 0x80;

            for (int i=8;i>0;i--)
                bMsg[sizeMsgBuff - i] = (byte)(sizeMsg >> ((8 - i) * 8) &
0x00000000000000ff);

            uint N = (uint)(bMsg.Length * 8) / 64;
```

```
            //iterasi sebanyak 3 kali
            for (int iterate = 0; i<3; i++)
            {
                 for (uint i = 0; i < N / 16; i++)
                 {
                     uint[] X = new uint[32];
                     X.Initialize();

                     uint a = A;
                     uint b = B;
                     uint c = C;
                     uint d = D;
                               uint e = E;
                     uint f = F;
                     uint g = G;
                     uint h = H;

                     CopyBlock(bMsg, X, i);
                     MD5Transformation(X, ref a, ref b, ref c, ref d, ref e, ref
      f, ref g, ref h);

                     A = A + a;
                     B = B + b;
                     C = C + c;
                     D = D + d;
                               E = E + e;
                     F = F + f;
                     G = G + g;
                     H = H + h;
                 }
            }

           String At = A.ToString("X8");
           String Bt = B.ToString("X8");
           String Ct = C.ToString("X8");
           String Dt = D.ToString("X8");
           String Et = E.ToString("X8");
           String Ft = F.ToString("X8");
           String Gt = G.ToString("X8");
           String Ht = H.ToString("X8");

           String hex = ReverseByte(A).ToString("X8") +
ReverseByte(B).ToString("X8") + ReverseByte(C).ToString("X8") +
ReverseByte(D).ToString("X8") + ReverseByte(E).ToString("X8") +
ReverseByte(F).ToString("X8") + ReverseByte(G).ToString("X8") +
ReverseByte(H).ToString("X8");
       }

       public void MD5Transformation(uint[] X,ref uint a,ref uint b,ref uint c,ref
uint d,ref uint e,ref uint f,ref uint g,ref uint h)
       {
                uint T[64];

                //Hitung tabel nilai radian dari 1 s/d 64
                for (int i=0;i<64;i++)
                {
                      T[i] := Math.floor(Math.abs(Math.sin(i + 1)) ×
(Math.pow(2,64)))
                }
```

```
/* Round 1 */
FF(ref a, b, c, d, e, f, g, h, X[0], s[0], T[0]); /* 1 */
FF(ref h, a, b, c, d, e, f, g, X[1], s[1], T[1]); /* 2 */
FF(ref g, h, a, b, c, d, e, f, X[2], s[2], T[2]); /* 3 */
FF(ref f, g, h, a, b, c, d, e, X[3], s[3], T[3]); /* 4 */
FF(ref e, f, g, h, a, b, c, d, X[4], s[4], T[4]); /* 5 */
FF(ref d, e, f, g, h, a, b, c, X[5], s[5], T[5]); /* 6 */
FF(ref c, d, e, f, g, h, a, b, X[6], s[6], T[6]); /* 7 */
FF(ref b, c, d, e, f, g, h, a, X[7], s[7], T[7]); /* 8 */
FF(ref a, b, c, d, e, f, g, h, X[8], s[8], T[8]); /* 9 */
FF(ref h, a, b, c, d, e, f, g, X[9], s[9], T[9]); /* 10 */
FF(ref g, h, a, b, c, d, e, f, X[10], s[10], T[10]); /* 11 */
FF(ref f, g, h, a, b, c, d, e, X[11], s[11], T[11]); /* 12 */
FF(ref e, f, g, h, a, b, c, d, X[12], s[12], T[12]); /* 13 */
FF(ref d, e, f, g, h, a, b, c, X[13], s[13], T[13]); /* 14 */
FF(ref c, d, e, f, g, h, a, b, X[14], s[14], T[14]); /* 15 */
FF(ref b, c, d, e, f, g, h, a, X[15], s[15], T[15]); /* 16 */

/* Round 2 */
GG(ref a, b, c, d, e, f, g, h, X[1], s[16], T[16]); /* 17 */
GG(ref h, a, b, c, d, e, f, g, X[6], s[17], T[17]); /* 18 */
GG(ref g, h, a, b, c, d, e, f, X[11], s[18], T[18]); /* 19 */
GG(ref f, g, h, a, b, c, d, e, X[0], s[19], T[19]); /* 20 */
GG(ref e, f, g, h, a, b, c, d, X[5], s[20], T[20]); /* 21 */
GG(ref d, e, f, g, h, a, b, c, X[10], s[21], T[21]); /* 22 */
GG(ref c, d, e, f, g, h, a, b, X[15], s[22], T[22]); /* 23 */
GG(ref b, c, d, e, f, g, h, a, X[4], s[23], T[23]); /* 24 */
GG(ref a, b, c, d, e, f, g, h, X[9], s[24], T[24]); /* 25 */
GG(ref h, a, b, c, d, e, f, g, X[14], s[25], T[25]); /* 26 */
GG(ref g, h, a, b, c, d, e, f, X[3], s[26], T[26]); /* 27 */
GG(ref f, g, h, a, b, c, d, e, X[8], s[27], T[27]); /* 28 */
GG(ref e, f, g, h, a, b, c, d, X[13], s[28], T[28]); /* 29 */
GG(ref d, e, f, g, h, a, b, c, X[2], s[29], T[29]); /* 30 */
GG(ref c, d, e, f, g, h, a, b, X[7], s[30], T[30]); /* 31 */
GG(ref b, c, d, e, f, g, h, a, X[12], s[31], T[31]); /* 32 */

/* Round 3 */
HH(ref a, b, c, d, e, f, g, h, X[5], s[32], T[32]); /* 33 */
HH(ref h, a, b, c, d, e, f, g, X[8], s[33], T[33]); /* 34 */
HH(ref g, h, a, b, c, d, e, f, X[11], s[34], T[34]); /* 35 */
HH(ref f, g, h, a, b, c, d, e, X[14], s[35], T[35]); /* 36 */
HH(ref e, f, g, h, a, b, c, d, X[1], s[36], T[36]); /* 37 */
HH(ref d, e, f, g, h, a, b, c, X[4], s[37], T[37]); /* 38 */
HH(ref c, d, e, f, g, h, a, b, X[7], s[38], T[38]); /* 39 */
HH(ref b, c, d, e, f, g, h, a, X[10], s[39], T[39]); /* 40 */
HH(ref a, b, c, d, e, f, g, h, X[13], s[40], T[40]); /* 41 */
HH(ref h, a, b, c, d, e, f, g, X[0], s[41], T[41]); /* 42 */
HH(ref g, h, a, b, c, d, e, f, X[3], s[42], T[42]); /* 43 */
HH(ref f, g, h, a, b, c, d, e, X[6], s[43], T[43]); /* 44 */
HH(ref e, f, g, h, a, b, c, d, X[9], s[44], T[44]); /* 45 */
HH(ref d, e, f, g, h, a, b, c, X[12], s[45], T[45]); /* 46 */
HH(ref c, d, e, f, g, h, a, b, X[15], s[46], T[46]); /* 47 */
HH(ref b, c, d, e, f, g, h, a, X[2], s[47], T[47]); /* 48 */

/* Round 4 */
II(ref a, b, c, d, e, f, g, h, X[0], s[48], T[48]); /* 49 */
II(ref h, a, b, c, d, e, f, g, X[7], s[49], T[49]); /* 50 */
II(ref g, h, a, b, c, d, e, f, X[14], s[50], T[50]); /* 51 */
II(ref f, g, h, a, b, c, d, e, X[5], s[51], T[51]); /* 52 */
II(ref e, f, g, h, a, b, c, d, X[12], s[52], T[52]); /* 53 */
II(ref d, e, f, g, h, a, b, c, X[3], s[53], T[53]); /* 54 */
II(ref c, d, e, f, g, h, a, b, X[10], s[54], T[54]); /* 55 */
II(ref b, c, d, e, f, g, h, a, X[1], s[55], T[55]); /* 56 */
II(ref a, b, c, d, e, f, g, h, X[8], s[56], T[56]); /* 57 */
II(ref h, a, b, c, d, e, f, g, X[15], s[57], T[57]); /* 58 */
II(ref g, h, a, b, c, d, e, f, X[6], s[58], T[58]); /* 59 */
II(ref f, g, h, a, b, c, d, e, X[13], s[59], T[59]); /* 60 */
II(ref e, f, g, h, a, b, c, d, X[4], s[60], T[60]); /* 61 */
```

```
            II(ref d, e, f, g, h, a, b, c, X[11], s[61], T[61]); /* 62 */
            II(ref c, d, e, f, g, h, a, b, X[2], s[62], T[62]); /* 63 */
            II(ref b, c, d, e, f, g, h, a, X[9], s[63], T[63]); /* 64 */
        }

        public uint leftRotate(uint x, uint y)
        {
            return ((x << (int) y) | (x >> (int) (64-y)));
        }

        public uint F(uint a, uint b, uint c)
        {
            return (a & b) | (~a & c);
        }

        public uint G(uint a, uint b, uint c)
        {
            return (a & c) | (b & ~c);
        }

        public uint H(uint a, uint b, uint c)
        {
            return a ^ b ^ c;
        }

        public uint I(uint a, uint b, uint c)
        {
            return b ^ (a | ~c);
        }

        public void FF(ref uint a, uint b, uint c, uint d, uint e, uint f, uint g,
uint h, uint x, uint s, uint ac)
        {
            a += F(b, c, d) + x + ac;
            a = leftRotate(a, s) + b;
                    e += F(f, g, h) + x + eg;
            e = leftRotate(e, s) + f;
        }

        public void GG(ref uint a, uint b, uint c, uint d, uint e, uint f, uint g,
uint h, uint x, uint s, uint ac)
        {
            a += G(b, c, d) + x + ac;
            a = leftRotate(a, s) + b;
                    e += G(f, g, h) + x + eg;
            e = leftRotate(e, s) + f;
        }

        public void HH(ref uint a, uint b, uint c, uint d, uint e, uint f, uint g,
uint h, uint x, uint s, uint ac)
        {
            a += H(b, c, d) + x + ac;
            a = leftRotate(a, s) + b;
                    e += H(f, g, h) + x + eg;
            e = leftRotate(e, s) + f;
        }

        public void II(ref uint a, uint b, uint c, uint d, uint e, uint f, uint g,
uint h, uint x, uint s, uint ac)
        {
            a += I(b, c, d) + x + ac;
            a = leftRotate(a, s) + b;
                    e += I(f, g, h) + x + eg;
            e = leftRotate(e, s) + f;
        }
```

```
        public void CopyBlock(byte[] bMsg, uint[] X, uint block)
        {
            block = block << 6;
            for (uint j = 0; j < 61; j += 4)
            {
                X[j >> 2] = (((uint)bMsg[block + (j + 3)]) << 24) |
                        (((uint)bMsg[block + (j + 2)]) << 16) |
                        (((uint)bMsg[block + (j + 1)]) << 8) |
                        (((uint)bMsg[block + (j)]));
            }
        }


        public uint ReverseByte(uint x)
        {
            return (((x & 0x000000ff) << 24) | (x >> 24) | ((x & 0x00ff0000) >> 8)
| ((x & 0x0000ff00) << 8));
        }
    }
}
```