Modifikasi Proses SHA-1 Berdasarkan Pembangkitan Bilangan Acak

Austin Dini Gusli - 13506101
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
austindini@itb.ac.id

***Abstrak*—Keamanan informasi semakin dituntut pada zaman sekarang. Pertukaran data dilakukan melalui berbagai media, baik yang sangat terjamin keamanannya hingga yang tidak terjamin sama sekali. Untuk menjamin keamanan informasi, tanda tangan digital dapat dimanfaatkan. Salah satu tanda tangan digital yang paling umum digunakan adalah SHA-1. Fungsi SHA-1 ini tentu telah mengalami banyak perkembangan. Makalah ini menawarkan dua buah modifikasi pada proses SHA-1 yang dapat diterapkan dengan cukup mudah. Modifikasi yang pertama adalah dengan membangkitkan sebuah bilangan acak sedangkan modifikasi yang kedua adalah dengan menambahkan panjang pesan asli pada *message digest*. Dengan demikian, modifikasi ini diharapkan untuk dikembangkan lebih lanjut sehingga SHA-1 dapat memperoleh jaminan keamanan yang lebih kuat.**

***Kata Kunci*—keamanan informasi, tanda tangan digital, SHA-1, message digest.**

I. PENDAHULUAN

Informasi menjadi salah satu kunci perkembangan hidup manusia. Kini, pertukaran informasi dapat terjadi di mana saja, kapan saja, dan melalui berbagai media. Media-media tersebut harus menjamin keamanan informasi yang dipertukarkan. Keamanan informasi dapat didukung oleh empat faktor, yaitu kerahasiaan pesan, otentikasi, keaslian pesan, dan anti-penyangkalan. Tiga di antara empat faktor tersebut dapat diselesaikan oleh tanda tangan digital.

Tanda tangan digital bergantung pada isi pesan dan kunci. Oleh karena itu, dokumen yang berbeda hampir pasti menghasilkan tanda tangan yang berbeda juga. Pemberian tanda tangan digital pada sebuah dokumen dapat dilakukan dengan mengenkripsi pesan atau menggunakan fungsi *hash* dan kriptografi kunci-publik.

Contoh pemanfaatan tanda tangan digital yang sangat sering ditemui adalah pada distribusi dokumen riset atau aplikasi. Pendistribusian dapat dilakukan dengan sangat mudah, khususnya melalui *web*. Pengubahan dokumen pun dapat dilakukan dengan mudah oleh sekelompok orang pula. Di sisi lain, fungsi *hash* merupakan sebuah fungsi yang dapat emnghasikan nilai yang sangat berbeda meskipun masukannya hanya berbeda satu bit. Dengan demikian, fungsi *hash* merupakan salah satu solusi terbaik untuk distribusi dokumen riset dan aplikasi.

Selain kelebihan tersebut, fungsi *hash* juga memiliki kelebihan lain, yakni tanda tangan yang dihasilkan akan selalu memiliki panjang yang tetap. Panjang tanda tangan ini tidak memakan memori besar, hanya 32-bit. Tanda tangan ini dapat ditempelkan langsung pada dokumen ataupun disimpan dokumen terpisah lainnya. Jadi, dokumen yang memakan memori besar pun masih dapat dijamin keasliannya oleh sebuah tanda tangan yang singkat dan tidak harus ditempelkan pada dokumen.

II. SHA-1

A. SHA sebagai fungsi hash

SHA (*Secure Hash Algorithm*) adalah sebuah standar fungsi *hash* satu arah. Fungsi *hash* satu arah adalah fungsi yang bekerja dalam satu arah; menghasilkan string yang tidak dapat diubah kembali menjadi pesan asli. SHA telah mengalami banyak perkembangan, dimulai dari rilisnya SHA-0 pada tahun 1993 hingga dipublikasikannya SHA-512. Namun, fungsi *hash* SHA yang paling umum digunakan adalah SHA-1 yang telah diimplementasikan di dalam berbagai aplikasi dan protokol keamanan seperti TSS, SSL, PGP, SSH, S/MIME, dan Ipsec.

Keluaran fungsi *hash* sering disebut sebagai nilai *hash* atau *message digest* (MD). Berikut merupakan beberapa contoh keluaran fungsi SHA-1.

Tabel 1 Contoh *message digest* fungsi SHA-1

Input	*Message Digest*
Keluaran fungsi hash.	0c51eddeac6f352aa9ff53d230c866a736e67011
Keluaran fungsi hash!	4cdfc4730a95127fcdb8f3ca9746300b71427aa8
Keluaran fungsi hash	3b531aa1a78fa48a3d9d77da4aa4d4e63c0c812c
Keluaran	19e1039427aac9c59aea8619d0c66545eb5ebdf6

Fungsi *hash* satu arah harus memenuhi sejumlah kriteria berikut.

a. *Preimage resistant*, yaitu tidak mungkin menemukan pesan masukan berdasarkan sebuah *message digest*.
b. *Second preimage resistant*, yaitu tidak mungkin menemukan dua masukan berbeda yang dapat

menghasilkan *message digest* yang sama.

c. *Collision resistant*, yaitu tidak mungkin menemukan dua pesan masukan dengan nilai *hash* yang sama.
d. Fungsi *hash* mudah dihitung.
e. Panjang *message digest* tetap.
f. Fungsi *hash* dapat diterapkan paada pesan masukan dengan panjang sebarang.

Meskipun fungsi *hash* satu arah harus memenuhi enam kriteria tersebut, masih dapat ditemukan sejumlah kolisi pada fungsi *hash*. Tabel berikut menggambarkan kolisi yang mungkin terjadi secara lebih detil.

Tabel 2 Kolisi algoritma fungsi *hash*

Algoritma	Ukuran MD (bit)	Ukuran Blok Pesan (bit)	Kolisi
MD2	128	128	Ya
MD4	128	512	Hampir
MD5	128	512	Ya
SHA-0	160	512	Ya
SHA-1	160	512	Hampir
SHA-256/224	256/224	512	Tidak
SHA-512/384	512/384	1024	Tidak
WHIRLPOOL	512	512	Tidak

Penggunaan fungsi *hash* sangat mudah dijumpai, sebagai contoh adalah pencocokan kunci antara *client* dan *server*. Sistem akan membandingkan nilai *hash* kunci yang dimasukkan pengguna dengan nilai *hash* yang tersimpan pada *server*. Jadi, informasi yang dipertukarkan tidak berupa pesan asli.

B. Algoritma SHA-1

Seperti yang dapat dilihat pada tabel, SHA-1 merupakan sebuah fungsi *hash* yang menghasilkan *message digest* sepanjang 160 bit. Tingkat keamanannya pun dapat dikatakan terjamin karena kolisi susah diperoleh.

Algoritma utama SHA-1 terdiri dari enam proses utama berikut.

a. Inisialisasi variabel kunci
b. Penambahan bit 1
c. Pemecahan pesan ke dalam kelompok berukuran 512-bit
d. Ekstensi pesan
e. Iterasi utama
f. Pembangkitan hash value.

Berikut merupakan *pseudocode* yang dapat lebih menggambarkan proses yang terjadi dalam algoritma SHA-1.

```
Initialize variables:
h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
h4 = 0xC3D2E1F0

Pre-processing:
append the bit '1' to the message
append 0 ≤ k < 512 bits '0', so that the
resulting message length (in bits)
   is congruent to 448 ≡ -64 (mod 512)
append length of message (before pre-
processing), in bits, as 64-bit big-endian
integer

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

    Extend the sixteen 32-bit words into
eighty 32-bit words:
    for i from 16 to 79
        w[i] = (w[i-3] xor w[i-8] xor w[i-
14] xor w[i-16]) leftrotate 1

    Initialize hash value for this chunk:
    a = h0
    b = h1
    c = h2
    d = h3
    e = h4

    Main loop:
    for i from 0 to 79
        if 0 ≤ i ≤ 19 then
            f = (b and c) or ((not b) and
d)
            k = 0x5A827999
        else if 20 ≤ i ≤ 39
            f = b xor c xor d
            k = 0x6ED9EBA1
        else if 40 ≤ i ≤ 59
            f = (b and c) or (b and d) or
(c and d)
            k = 0x8F1BBCDC
        else if 60 ≤ i ≤ 79
            f = b xor c xor d
            k = 0xCA62C1D6

        temp = (a leftrotate 5) + f + e + k
+ w[i]
        e = d
        d = c
        c = b leftrotate 30
        b = a
        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

Produce the final hash value (big-endian):
digest = hash = h0 append h1 append h2
append h3 append h4
```

Gambar 1 *Pseudocode* SHA-1 asli

Pada tahun 2005, Rijmen dan Oswald mempublikasikan kolisi yang mungkin terjadi pada SHA-1 tereduksi (hanya menggunakan 53 dari 80 putaran) dengan kompleksitas sekitar 2^{80} operasi. Juga pada tahun

2005, Xiayoun Wang, Yiqun Lisa Yin, dan Hongbo Yo mempublikasikan kolisi pada versi penuh SHA-1 dengan tingkat kompleksitas 2^{69} operasi. Kompleksitas tersebut masih tergolong sebagai kompleksitas yang besar sehingga SHA-1 masih dapat dipercaya sebagai salah satu fungsi *hash* yang paling dapat diandalkan.

III. MODIFIKASI ALGORITMA SHA-1

A. *Alternatif nilai* `f`

Seperti yang telah dijelaskan sebelumya, SHA-1 telah mengalami banyak perkembangan. Terdapat sejumlah alternatif yang dapat digunakan untuk pemberian nilai `f` pada proses *`main loop`* dalam *pseudocode* di atas. Contoh alternatif yang ada untuk pemberian nilai *`f`* bagi kelompok 0 – 19 antara lain:

```
f = (b and c) or ((not b) and d).....(1)
f = d xor (b and (c xor d))..........(2)
f = (b and c) xor ((not b) and d)....(3)
f = (b and c) + ((not b) and d)......(4)
```

Sedangkan contoh alternatif yang ada untuk pemberian nilai `f` untuk kelompok 40 hingga 59.

```
f = (b and c)or(b and d)or(c and d)..(5)
f = (b and c) or (d and (b or c))....(6)
f = (b and c) or (d and (b xor c))...(7)
f = (b and c) + (d and (b xor c))....(8)
f = (b and c) xor (b and d) xor (c and d)
      ..............................(9)
```

Umumnya, fungsi *hash* telah langsung menentukan metode pengisian nilai `f` pada *code*. Makalah ini mengajukan sebuah metode untuk membangkitkan alternatif yang digunakan oleh aplikasi. Alternatif yang dipakai dibangkitkan berdasarkan angka yang diperoleh dari penjumlahan bit semua karakter dokumen dan jumlah karakter dalam pesan, selanjutnya nilai tersebut modulo bit karakter pada tengah dokumen dan akhirnya modulo jumlah alternatif yang ada. Berikut merupakan *pseudocode* pembangkitan bilang acak tersebut.

```
int sum ← 0
int chars ← 0
for each character in message
    sum ← sum + toInt(character)
    chars ← chars + 1
int alterNum ←
    ((random(chars) % character[chars/2])
     % alternativesNum)
alternative-f(message, alterNum)
```

Gambar 2 *Pseudocode* pembangkitan bilangan acak

Pseudocode pembangkitan nilai random tersebut dapat divariasikan sesuai keinginan pengembang. Dalam pembuatan makalah ini, metode tersebut dipilih karena dianggap dapat menghasilkan bilangan acak yang dulit diprediksi karena telah memanfaatkan fungsi random yang tersedia. Selain itu, bilangan pembangkit bilangan acak dapat diperoleh dengan cara yang mudah (tidak membutuhkan komputasi yang berat).

Jumlah alternatif yang digunakan pun masih sangat minim, hanya empat buah untuk kelompok iterasi 0 hingga 19 dan lima buah untuk kelompok iterasi 40 hingga 59. Jumlah alternatif yang digunakan bukan fokus makalah ini, melainkan cara penggunaan alternatif ini.

B. *Penyertaan panjang pesan asli*

Penempelan panjang pesan asli pada MD akan menghasilkan nilai MD yang unik. Hal ini dikarenakan tidak ada dua pesan berbeda dengan panjang yang sama dapat menghasilkan MD yang sama. Dengan kata lain, *hash*(pesan1) ≠ *hash*(pesan2) akan selalu berlaku jika dengan *panjang*(pesan1) = *panjang*(pesan2) dan pesan1 ≠ pesan2.

Panjang pesan asli yang dapat ditampung ini sebaiknya disesuaikan dengan perkiraan panjang pesan maksimal. Jadi, penempelan panjang pesan asli akan menggunakan setidaknya 10 karakter heksa untuk pesan sepanjang 5×10^6 karakter. Dengan demikian, modifikasi ini akan bergantung pada asumsi jumlah karakter pada dokumen.

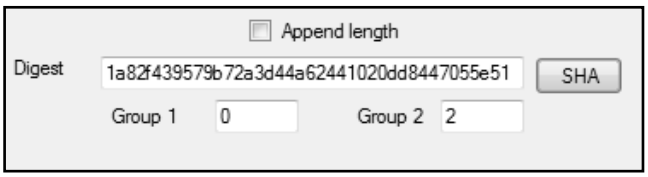
Karena fungsi *hash* akan menghasilkan pesan dengan panjang yang tetap, *message digest* dengan panjang yang berbeda dengan panjang pada umumnya akan memicu kecurigaan. Oleh karena itu, *message digest* yang ditambahkan panjang dokumen sebaiknya menjadi input fungsi *hash* lagi. *Message digest* hasil fungsi tersebut yang menjadi *message digest* sebenarnya.

Penerapan fungsi *hash* sebanyak dua kali tentu menambah kompleksitas. Kompleksitas yang ada pun tidak banyak berubah sebab panjang dokumen yang akan mengalami *hash* hanya 200 bit (asumsi: panjang pesan asli maksimal sebesar 10 karakter heksa). Di sisi lain, cara ini mampu memberikan fungsi otentikasi yang lebih baik.

IV. IMPELEMENTASI DAN PENGUJIAN

A. *Implementasi*

Implementasi modifikasi yang dipaparkan pada bagian sebelumnya dilakukan pada IDE Microsoft Visual Studio 2008 dan bahasa pemrograman C#. Antarmuka aplikasi yang dikembangkan dapat dilihat pada gambar di bawah ini.



Gambar 3 Antarmuka program yang diimplementasikan

Berikut merupakan potongan kode yang digunakan untuk membangkitkan bilangan acak.

```
public int randomGroup1()
{
     Random random = new Random(charsNum);
     return random.Next() % medianChar % 4;
}

public int randomGroup2()
{
     Random random = new Random(charsNum);
     random.Next();
     return random.Next() % medianChar % 5;
}
```

Gambar 4 Kode pembangkit bilangan acak

Gambar berikut merupakan potongan kode SHA-1 sesungguhnya, tidak memanfaatkan pembangkitan nilai acak di atas.

```
//pra main-loop

for (i = 0; i < 80; i++)
{
   if ((i <= 19) && (i >= 0))
   {
      f = (b & c) | ((~b) & d);
      k = 0x5A827999;
   }
   else if ((i <= 39) && (i >= 20))
   {
      f = b ^ c ^ d;
      k = 0x6ED9EBA1;
   }
   else if ((i <= 59) && (i >= 40))
   {
      f = (b & c) ^ (b & d) ^ (c & d);
      k = 0x8F1BBCDC;
   }
   else if ((i <= 79) && (i >= 60))
   {
      f = b ^ c ^ d;
      k = 0xCA62C1D6;
   }
   temp = RotateLeft(a, 5) + f + e + k +
hextemp2[i];
   e = d;
   d = c;
   c = RotateLeft(b, 30);
   b = a;
   a = temp;

}

//pasca main-loop
```

Gambar 5 Kode asli SHA-1

Sedangkan berikut merupakan kode yang telah memanfaatkan nilai acak yang di-generate dengan fungsi randomGroup1 dan randomGroup2 di atas. Variabel `random1` menampung nilai dari fungsi `randomGroup1` dan variabel `random2` menampung nilai dari fungsi `randomGroup2`.

```
//pra main-loop

for (i = 0; i < 80; i++)
{
   if ((i <= 19) && (i >= 0))
   {
      if (random1 == 0)
         f = (b & c) | ((~b) & d);
         //(b and c) or ((not b) and d)
      else if (random1 == 1)
         f = d ^ (b & (c ^ d));
         //d xor (b and (c xor d))
      else if (random1 == 2)
         f = (b & c) ^ ((~b) & d);
        //(b and c) xor ((not b) and d)
      else
         f = (b & c) + ((~b) & d);
        //(b and c) + ((not b) and d)
      k = 0x5A827999;
   }
   else if ((i <= 39) && (i >= 20))
   {
     f = b ^ c ^ d;
     k = 0x6ED9EBA1;
   }
   else if ((i <= 59) && (i >= 40))
   {
      if (random2 == 0)
         f = (b & c) ^ (b & d) ^ (c & d);
         //(b and c)xor(b and d)xor(c and
d)
      else if (random2 == 1)
         f = (b & c) | (d & (b | d));
         //(b and c) or (d and (b or c))
      else if (random2 == 2)
       f = (b & c) | (d & (b ^ d));
    //(b and c) or (d and (b xor c))
    else if (random2 == 3)
       f = (b & c) + (d & (b ^ d));
      //(b and c) + (d and (b xor c))
    else
       f = (b & c) | (b & d) | (c & d);
      //(b and c) or (b and d) or (c and d)
    k = 0x8F1BBCDC;
  }
  else if ((i <= 79) && (i >= 60))
  {
    f = b ^ c ^ d;
    k = 0xCA62C1D6;
  }
  temp = RotateLeft(a, 5) + f + e + k +
hextemp2[i];
  e = d;
  d = c;
  c = RotateLeft(b, 30);
  b = a;
  a = temp;
}

//pasca main-loop
```

Gambar 6 Modifikasi kode SHA-1

Berikut adalah gambar implementasi fungsi pengubahan panjang dokumen menjadi heksa. Fungsi ini digunakan untuk modifikasi B, yaitu penyertaan panjang pesan asli pada ujung *message digest*.

```
public String lengthToHexa()
{
   int copy = charsNum;
   String result = "";
   do
   {
      if (copy % 16 < 10)
         result = result.Insert(0, ((copy %
16)).ToString());
```

```
      else
         result =
result.Insert(0,((char)(copy % 16 +
87)).ToString());
      copy = copy / 16;
    } while (copy > 0);
    while (result.Length < 10)
    {
       result = result.Insert(0, "0");
    }
    return result;
}
```

Gambar 7 Kode pengubah integer ke heksa

Output fungsi tersebut ditambahkan ke akhir *message digest* fungsi *hash* pertama. Selanjutnya, *message digest* ini bertambah 10 karakter heksa. *Message digest* ini kemudian ditandantangani kembali. Hasil penandatanganan ini merupakan hasil akhir.

```
digest =
   ConvertToHexString(sha.getH0()) +
   ConvertToHexString(sha.getH1()) +
   ConvertToHexString(sha.getH2()) +
   ConvertToHexString(sha.getH3()) +
   ConvertToHexString(sha.getH4()) +
   io.lengthToHexa();
```

Gambar 8 Pembangkitan *hash value*

B. *Pengujian*

Pengujian dilakukan dengan menggunakan masukan yang sama dengan masukan pada tabel. Berikut merupakan hasil pengujian yang dilakukan pada aplikasi yang telah menerapkan dua modifikasi yang telah dijelaskan pada bagian III.

Tabel 3Hasil pengujian SHA-1 yang dimodifikasi

Input	***Message Digest* SHA-1**	***Message Digest* dengan nilai `f` alternatif**	***Message Digest* dengan nilai `f` alternatif dan penyertaan panjang pesan asli**
Keluaran fungsi hash.	0c51eddeac6f352aa9ff53d230c866a736e67011	0c51eddeac6f352aa9ff53d230c866a736e67011	1c3429623cf0b11dfd5acde4acbb0ccbd072a7a0
Keluaran fungsi hash!	4cdfc4730a95127fcdb8f3ca9746300b71427aa8	4cdfc4730a95127fcdb8f3ca9746300b71427aa8	6de987344c12fd6794a59ea27faa0e6d06aaf2fc
Keluaran fungsi hash	3b531aa1a78fa48a3d9d77da4aa4d4e63c0c812c	c47a97a0706ece045d79215d15431ac5bbd28f57	e2c13770adf1cae0d5f0e207e2047325601eb6b1
Keluaran	19e1039427aac9c59aea8619d0c66545eb5ebdf6	1a82f439579b72a3d44a62441020dd8447055e51	4822cee5f5420aa423c72c540ab978e9a4b518ca

Hasil pengujian tersebut menunjukkan bahwa modifikasi dengan memberi nilai `f` secara acak dapat menghasilkan *message digest* yang sama dengan SHA-1 asli. Hal ini berarti bilangan acak yang dibangkitkan menentukan bahwa nilai `f` akan diisi dengan cara yang sama seperti pada fungsi SHA-1. Penyertaan panjang pesan asli pada *message digest* hasil `f` tentu akan memberikan panjang pesan yang sama dengan SHA-1 murni, yaitu 160-bit.
Meskipun SHA-1 dapat mengalami kolisi, *avalanche effect* yang memberi jaminan keamanan yang kuat.

V. PENUTUP

Berdasarkan pemaparan yang ada pada bagian-bagian sebelumnya, dapat diperoleh sejumlah kesimpulan berikut.

a. Fungsi *hash* satu arah merupakan salah satu cara pemberian tanda tangan yang dapat dianggap sangat aman.
b. Modifikasi dengan mengisi nilai `f` sesuai hasil pembangkitan bilangan acak dapat menghasilkan *message digest* yang sama dengan SHA-1 asli ataupun berbeda.
c. Modifikasi penyertaan panjang pesan asli dapat meningkatkan jaminan keaslian data, namun tidak menutup kemungkinan adanya kolisi.

Hasil riset yang telah dilakukan belum melalui sejumlah tahap pengujian, khususnya pengujian serangan. Modifikasi ini sebaiknya dikembangkan lebih lanjut, khususnya untuk mencari tahu apakah modifikasi ini mempermudah kolisi atau tidak.

DAFTAR REFERENSI

[1] R. Munir, *Bahan Kuliah IF5054 Kriptografi*. Departemen Teknik Informatika, Institut Teknologi Bandung, 2004.
[2] V. Rijmen, E. Oswald, “Update on SHA-1”, 2005.
[3] D. Eastlake, P. Jones, Request For Comment (RFC)-3174 : US Secure Hash Algorithm 1 (SHA1). The Internet Society. (http://tools.ietf.org/html/rfc3174), 2001, tanggal akses : 17 Mei 2010

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 17 Mei 2010

Austin Dini Gusli - 13506101