IMPLEMENTASI DAN PERBANDINGAN PERFORMA ALGORITMA HASH SHA-1, SHA-256, DAN SHA-512

Amudi Sebastian – NIM : 13503017

Program Studi Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
E-mail: if13017@students.if.itb.ac.id

Abstrak

Dua tujuan terpenting dari sistem sekuriti adalah kerahasiaan dan integritas pesan. Namun, ada kalanya dimana kerahasiaan tidak menjadi fokus utama, tapi kita tetap menginginkan agar pesan yang dikirim melalui media internet tidak diubah oleh pihak lain. Dalam hal ini, fungsi *hash* dapat membantu dalam menjaga integritas pesan. Fungsi *hash* dapat menghitung nilai *hash* atau *message digest* dari pesan apapun yang diberikan sebagai input. Nilai *message digest* yang dihasilkan tidak dapat dikonstruksi kembali menjadi pesan semula, sehingga fungsi *hash* disebut juga fungsi searah. Keamanan dan integritas pesan dapat diverifikasi dengan fungsi *hash* ini. Ada berbagai jenis algoritma *hash* yang telah dibuat, diantaranya memiliki kelebihan dan kekurangan, baik dari sisi implementasi, kinerja, serta ukuran *message digest* yang dihasilkan. Makalah ini akan mencoba membandingkan tiga jenis algoritma *hash* yang telah distandardisasi oleh National Institute of Standards and Technology. Algoritma yang telah distandardisasi diantaranya SHA-1 dan SHA-2 (SHA-256 dan SHA-512). Implementasi dilakukan dalam bahasa C++. Variabel yang diuji dan dibandingkan adalah kinerja algoritma untuk data kecil (satu block) dan data besar, jumlah siklus yang harus dilalui per byte pesan, serta tingkat keamanan dari nilai *message digest* yang dihasilkan.

***Kata kunci**: Secure Hash Algorithm, SHA, message digest, hash*

1. Pendahuluan

Perkembangan teknologi informasi, terutama dalam bidang jaringan komunikasi dan internet, telah membawa perubahan besar pada kehidupan sehari-hari. Penyebaran data dan informasi menjadi lebih mudah karena adanya media internet untuk distribusi dan pertukaran data. Kerahasiaan dan integritas data yang dikirimkan, yang merupakan tujuan terpenting dalam sistem sekuriti, merupakan hal yang perlu diperhatikan dalam pertukaran data. Setiap data yang dikirimkan dapat terkena serangan, sehingga integritas data menjadi hilang.

Diperlukan suatu mekanisme untuk menjaga integritas data, yaitu memastikan data tidak diubah pihak lain dalam proses pengirimannya. Fungsi *hash* dapat digunakan untuk menghitung nilai *hash* dari suatu data yang akan dikirimkan. Pihak penerima dapat menggunakan nilai *hash* awal tersebut dan membandingkan dengan nilai *hash* pesan yang telah diterima untuk mengetahui apakah pesan telah dimanipulasi atau tidak. Dengan fungsi *hash*, pertukaran data melalui media internet dapat lebih terjamin, dimana data yang diterima dapat dipastikan masih asli dan tidak diubah oleh pihak lain. Pada makalah ini akan dibahas implementasi dan pengujian tiga fungsi *hash*, yaitu SHA-1, SHA-256, dan SHA-512.

2. Fungsi *Hash*

Fungsi *hash* memiliki sifat searah, sehingga sering disebut *one way hash function*. Sembarang pesan M berukuran bebas dikompresi oleh fungsi *hash* H melalui persamaan

$$h = H(M)$$

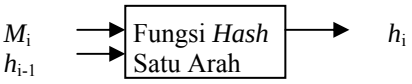
Sifat-sifat fungsi *hash* adalah sebagai berikut [1]:

1. Fungsi H dapat diterapkan pada blok data berukuran berapa saja.
2. H menghasilkan nilai (h) dengan panjang tetap (*fixed-length output*).
3. $H(x)$ mudah dihitung untuk setiap nilai x yang diberikan.
4. Untuk setiap h yang dihasilkan, tidak mungkin dikembalikan nilai x sedemikian sehingga $H(x) = h$. Itulah sebabnya fungsi H dikatakan fungsi *hash* satu-arah (*one-way hash function*).
5. Untuk setiap x yang diberikan, tidak mungkin mencari $y \neq x$ sedemikian sehingga $H(y) = H(x)$.

6. Tidak mungkin mencari pasangan x dan y sedemikian sehingga $H(x) = H(y)$.

Nilai fungsi *hash* satu arah biasanya berukuran kecil, sedangkan pesan berukuran besar.

Fungsi *hash* memiliki algoritma yang *iterative* dan searah, yang dapat memproses pesan yang diberikan untuk menghasilkan representasi yang lebih pendek yang disebut *message digest*. Untuk menghasilkan nilai *message digest* dari pesan besar, fungsi *hash* menerima masukan berupa isi blok pesan saat ini serta nilai *hash* dari blok pesan sebelumnya, seperti yang digambarkan seperti skema dibawah ini.



2.1. Standardisasi *Hash*

National Institute of Standards and Technology di Amerika Serikat melakukan standardisasi dalam fungsi *hash*, untuk meningkatkan keamanan dalam pertukaran data. Dalam dokumen FIPS 180-2 [2], ditetapkan lima jenis fungsi *hash*, yaitu SHA-1, dan SHA-2 yang terdiri dari SHA-256, SHA-224, SHA-512, SHA-384, yang masing-masing didefinisikan dan dijelaskan aturan-aturannya dalam dokument tersebut. Kelima fungsi *hash* tersebut mirip satu dengan yang lainnya, tetapi memiliki perbedaan yang mencolok juga. Misalnya adalah keluaran *message digest* yang dihasilkan tiap fungsi SHA diatas masing-masing berbeda satu dengan yang lainnya. Dengan adanya standardisasi tersebut, diharapkan komunikasi data dapat menjadi lebih aman.

2.2. Tingkat Keamanan

Dengan adanya standardisasi, tentunya diharapkan bahwa penggunaan fungsi *hash* semakin meluas dan tingkat keamanan dalam pertukaran data menjadi lebih tinggi.

Mengingat SHA adalah fungsi *hash* satu arah, maka hampir tidak mungkin (secara komputasional) untuk mengkonstruksi kembali pesan semula dari sebuah *message digest*. Lebih panjang *message digest*, serta lebih kompleksnya algoritma yang digunakan, maka makin sulit pula pencarian pesan semula yang bersesuaian dengan *message digest* tertentu. Kompleksitas algoritma dapat ditingkatkan dengan menambah jumlah *loop* yang dilakukan, menggunakan panjang data yang lebih panjang, serta menggunakan output *message digest* yang lebih panjang. Ketiga algoritma yang akan dibahas sampai saat ini masih dianggap aman. Untuk lebih lengkapnya, perbandingan keamanan dari ketiga algoritma tersebut akan dibahas pada bagian 4.3.

3. *Secure Hash Algorithm* (SHA)

Algoritma fungsi SHA dapat dideskripsikan dan dibagi menjadi dua bagian: *preprocessing* dan perhitungan *message digest*. *Preprocessing* melibatkan proses penambahan bit pengganjal (*padding bits*), membagi pesan menjadi blok-blok dengan panjang tertentu, serta mengeset nilai awal untuk digunakan pada perhitungan *message digest*. Pada proses perhitungan *message digest*, dilakukan proses pembangkitan *message schedule* dari pesan, dan kemudian *message schedule* tersebut, bersama dengan fungsi-fungsi lainnya serta konstanta-konstanta yang telah terdefinisi, digunakan secara iteratif untuk membangkitkan nilai *hash* akhir.

Ketiga algoritma SHA yang dibahas memiliki perbedaan utama pada jumlah bit sekuriti yang diberikan pada data yang akan di-*hash*, yang berhubungan langsung pada panjang *message digest* yang dihasilkan. Selain itu, perbedaan juga terdapat pada ukuran blok dan *word* data yang digunakan selama proses komputasi. Perbedaan ketiga varian algoritma SHA secara lengkap dapat dilihat pada **Tabel 1**.

Untuk mempermudah implementasi fungsi *hash*, maka perlu dibuat suatu interface umum yang seragam yang berguna dalam pengaksesan objek-objek instansiasi dari kelas fungsi *hash* tersebut. Dengan cara pengaksesan yang seragam, maka implementasi ketiga fungsi *hash* tersebut juga dapat dibuat semirip mungkin. Sehingga, ketiga jenis fungsi *hash* yang akan diimplementasi dapat dibandingkan secara langsung dalam segi performa.

Operasi-operasi atau *method* yang diperlukan pada penggunaan algoritma SHA secara umum adalah `Init()`, `Update()`, dan `Final()`. Fungsi `Init()` akan melakukan inisiasi variabel dengan konstanta-konstanta algoritma SHA yang bersangkutan. Fungsi `Update()` berguna untuk menambahkan sebuah *byte* ke dalam pesan yang akan diproses. Fungsi `Final()` berguna untuk mengakhiri proses komputasi SHA sekaligus menambahkan bit-bit pengganjal, serta mengembalikan *message digest* akhir yang dihasilkan. Selain itu, dibutuhkan juga fungsi `Transform()` yang digunakan secara internal, untuk melakukan kalkulasi nilai *hash* sesuai jenis algoritma SHA yang digunakan.

Oleh karena itu, untuk mengimplementasi fungsi SHA, dibutuhkan definisi kelas abstrak `IHash` dalam bahasa C++ yang akan digunakan sebagai *parent class* bagi algoritma SHA yang diimplementasi (SHA-1, SHA-256, SHA-512), yang mendeklarasikan fungsi-fungsi diatas. Fungsi-fungsi tersebut dideklarasikan sebagai virtual, dan akan direalisasi di kelas anak yang merupakan implementasi konkrit dari suatu algoritma SHA. Untuk lebih jelasnya, diagram kelas dapat dilihat pada **Gambar 1**.

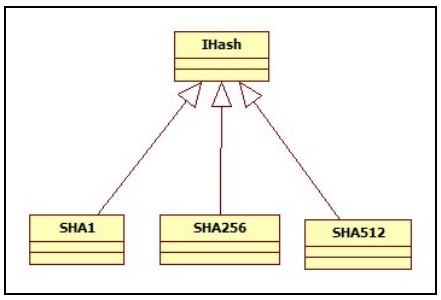
Selain itu, diperlukan juga tambahan variabel-variabel dan makro-makro yang berguna pada implementasi ketiga algoritma SHA secara keseluruhan, seperti variabel untuk menyimpan nilai *hash* sementara, variabel panjang pesan, makro yang mendefinisikan fungsi-fungsi SHA seperti `ROTR`, `ROTL`, `MAJ`, serta makro untuk konversi tipe data. Kelas `IHash` ini diimplementasi dalam file `IHash.h`. Berikut ini adalah definisi makro dalam kelas `IHash`.

```
#if !defined(_IHASH_H)
#define _IHASH_H

#include <string.h>

#define SHFR(x, n) (x >> n)
#define ROTR(x, n) (x >> n) | (x <<
((sizeof(x) << 3) - n)))
#define ROTL(x, n) ((x << n) | (x
>> ((sizeof(x) << 3) - n)))
#define CH(x, y, z) ((x & y) ^ (~x
& z))
#define MAJ(x, y, z) ((x & y) ^ (x
& z) ^ (y & z))

// makro konversi tipe data
#define UNPACK32(x, str) \
{                        \
*((str) + 3) = (unsigned char) ((x)
);           \
*((str) + 2) = (unsigned char) ((x)
>>  8);      \
*((str) + 1) = (unsigned char) ((x)
>> 16);      \
*((str) + 0) = (unsigned char) ((x)
>> 24);      \
}

#define PACK32(str, x) \
{                      \
*(x) = ((unsigned int) *((str) + 3)
)     \
| ((unsigned int) *((str) + 2) <<
8)     \
| ((unsigned int) *((str) + 1) <<
16)     \
| ((unsigned int) *((str) + 0) <<
24);    \
}
```

```
#define UNPACK64(x, str) \
{                        \
*((str) + 7) = (unsigned char) ((x)
);      \
*((str) + 6) = (unsigned char) ((x)
>>  8);      \
*((str) + 5) = (unsigned char) ((x)
>> 16);      \
*((str) + 4) = (unsigned char) ((x)
>> 24);      \
*((str) + 3) = (unsigned char) ((x)
>> 32);      \
*((str) + 2) = (unsigned char) ((x)
>> 40);      \
*((str) + 1) = (unsigned char) ((x)
>> 48);      \
*((str) + 0) = (unsigned char) ((x)
>> 56);      \
}

#define PACK64(str, x) \
{                      \
*(x) = ((unsigned long long)
*((str) + 7)      )     \
| ((unsigned long long) *((str) +
6) <<  8)     \
| ((unsigned long long) *((str) +
5) << 16)     \
| ((unsigned long long) *((str) +
4) << 24)     \
| ((unsigned long long) *((str) +
3) << 32)     \
| ((unsigned long long) *((str) +
2) << 40)     \
| ((unsigned long long) *((str) +
1) << 48)     \
| ((unsigned long long) *((str) +
0) << 56);    \
}
```

Dan selanjutnya, dibawah ini adalah definisi kelas abstrak `IHash` dengan fungsi-fungsi dan variabel yang dibutuhkan sesuai dengan yang telah dibahas diatas.

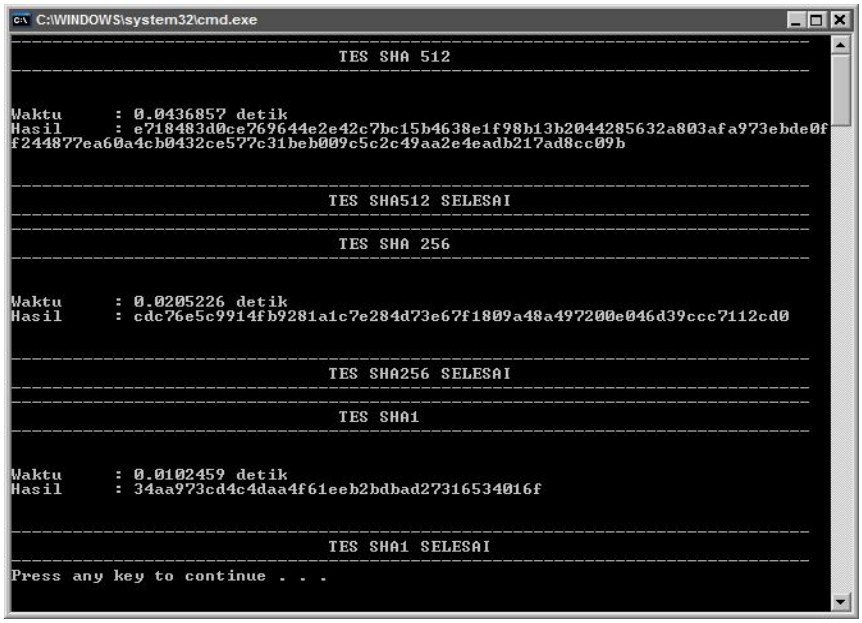
```
class IHash
{
public:
    virtual void Init() = 0;
    virtual void Update(unsigned
char* p_Message, unsigned int
p_Length) = 0;
    virtual void Final(unsigned
char* p_MessageDigest) = 0;
protected:
    virtual void Transform(unsigned
char* p_Message, unsigned int
p_BlockNum) = 0;
    unsigned char* m_Block;
    unsigned int m_TotalLength;
};

#endif  //_IHASH_H
```



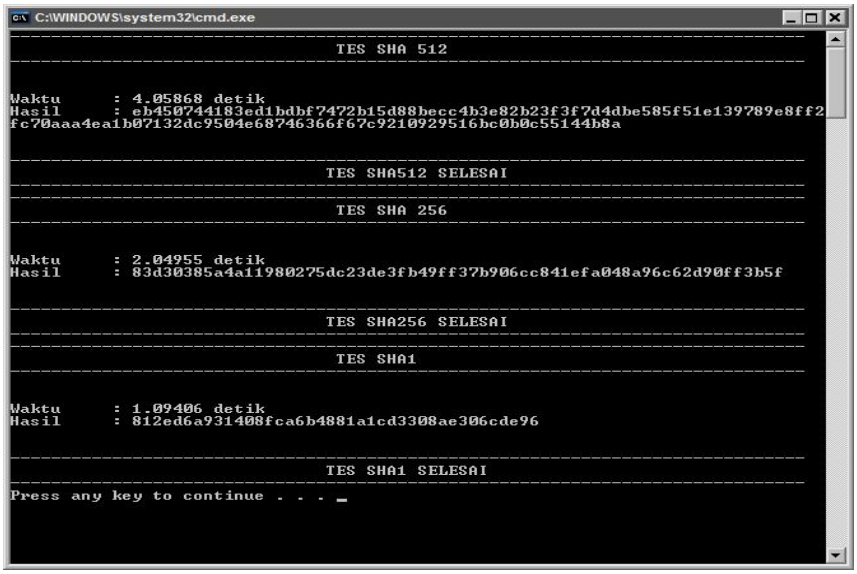
Gambar 1. Diagram Kelas

Algoritma	Ukuran Pesan (bit)	Ukuran Blok (bit)	Ukuran Word (bit)	Ukuran *Message Digest* (bit)	Bit Sekuriti (bit)
SHA-1	$<2^{64}$	512	32	160	80
SHA-256	$<2^{64}$	512	32	256	128
SHA-512	$<2^{128}$	1024	64	512	256

Tabel 1. Perbedaan Karakteristik Varian Algoritma SHA



Gambar 2. Perbandingan Performa SHA-1, SHA-256, dan SHA-512 dengan Optimasi Compiler



Gambar 3. Perbandingan Performa SHA-1, SHA-256, dan SHA-512 dengan Optimasi Compiler dan Data Sangat Besar

3.1. SHA-1

Algoritma SHA-1 dapat digunakan untuk menghitung nilai *message digest* dari sebuah pesan, dimana pesan tersebut memiliki panjang maksimum 2^{64} bit. Algoritma ini menggunakan sebuah *message schedule* yang terdiri dari 80 elemen 32-bit *word*, lima buah variabel 32-bit, dan variabel penyimpan nilai *hash* 5 buah *word* 32-bit. Hasil akhir dari algoritma SHA-1 adalah sebuah *message digest* sepanjang 160-bit.

Preprocessing dilakukan dengan menambahkan bit pengganjal, membagi-bagi pesan dalam block berukuran 512-bit, dan terakhir menginisiasi nilai *hash* awal. Misalkan panjang pesan *M* adalah *l* bit. Proses penambahan bit pengganjal adalah pertama tambahkan bit "1" pada akhir pesan, diikuti dengan bit "0" sejumlah *k*, dimana *k* adalah solusi dari persamaan:

$$l + 1 + k \equiv 448 \bmod 512$$

Kemudian tambahkan 64-bit block yang menyatakan panjang pesan semula (*l*) dalam representasi biner.

Proses komputasi SHA-1 dapat dilihat pada bagian implementasi dibawah.

3.1.1. Deskripsi

Dalam proses komputasinya, SHA-1 menggunakan sederetan fungsi lojik $f_0, f_1,\ldots,f_{79}$ dalam proses komputasinya. Setiap fungsi f_t, dimana $0 < t < 79$, beroperasi dengan tiga *word* berukuran 32-bit (*x*, *y*, dan *z*), dan menghasilkan keluaran berupa *word* 32-bit. Fungsi $f_t(x, y, z)$ didefinisikan sebagai berikut, masing-masing untuk $0 \le t \le 19$, $20 \le t \le 39$, $40 \le t \le 50$, dan $60 \le t \le 79$ [2]:

$$Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$$
$$Parity(x, y, z) = x \oplus y \oplus z$$
$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$
$$Parity(x, y, z) = x \oplus y \oplus z$$

Kemudian, untuk perhitungan *message digest* dengan SHA-1, digunakan konstanta sebagai berikut (dalam nilai *hexadecimal*) [3]:

```
K(t) = 5A827999 ( 0 ≤ t ≤ 19)
K(t) = 6ED9EBA1 (20 ≤ t ≤ 39)
K(t) = 8F1BBCDC (40 ≤ t ≤ 59)
K(t) = CA62C1D6 (60 ≤ t ≤ 79)
```

Nilai *hash* awal (*initial hash value*) dari algoritma SHA-1 adalah sebagai berikut, sesuai yang telah ditetapkan pada [2]:

```
H0[0] = 67452301
H0[1] = efcdab89
H0[2] = 98badcfe
H0[3] = 10325476
H0[4] = c3d2e1f0
```

3.1.2. Implementasi

Dalam implementasinya, kelas `SHA1` didefinisikan sebagai turunan kelas `IHash`. Selain itu, pada kelas `SHA1` didefinisikan juga dua buah makro, yaitu `SHA1_BLOCK_SIZE` dan `SHA1_DIGEST_SIZE`, yang menyatakan besar block dan *message digest* dalam `SHA1`. Berikut ini adalah deklarasi kelas `SHA1` (file `SHA1.h`).

```
#if !defined(_SHA1_H)
#define _SHA1_H

#include "IHash.h"

#define SHA1_BLOCK_SIZE (512 / 8)
#define SHA1_DIGEST_SIZE (160 / 8)

class SHA1 : public IHash
{
public:
    SHA1();
    virtual ~SHA1();
       void Init();
       void Update(unsigned char*
p_Message, unsigned int p_Length);
       void Final(unsigned char*
p_MessageDigest);
protected:
    void Transform(unsigned char*
p_Message, unsigned int
p_BlockNum);
private:
       unsigned int m_H[5];
       unsigned int m_H0[5];
       unsigned int m_K[4];
       unsigned char m_Length;
};

#endif  //_SHA1_H
```

Kemudian, seluruh fungsi yang dideklarasikan pada `SHA1.h` harus diimplementasi pada `SHA1.cpp`. Berikut ini adalah implementasi konstruktor kelas SHA1 yang menginisiasi nilai-nilai konstanta `m_H0` sesuai dengan aturan yang telah dibahas pada 3.1.1, dan mengalokasikan variabel `m_Block` yang menyimpan *block* pesan.

```
#include "SHA1.h"

SHA1::SHA1()
{
    m_Block = new unsigned char[2 *
SHA1_BLOCK_SIZE];
```

```
    // inisiasi H
    m_H0[0] = 0x67452301;
       m_H0[1] = 0xEFCDAB89;
       m_H0[2] = 0x98BADCFE;
       m_H0[3] = 0x10325476;
    m_H0[4] = 0xC3D2E1F0;

    // inisiasi K
    m_K[0] = 0x5A827999;
       m_K[1] = 0x6ED9EBA1;
       m_K[2] = 0x8F1BBCDC;
       m_K[3] = 0xCA62C1D6;
}
```

Kemudian, dibawah ini adalah implementasi destruktor yang mendealokasi memori yang telah dialokasi sebelumnya.

```
SHA1::~SHA1()
{
    delete m_Block;
}
```

Fungsi `Init()` dibawah ini melakukan inisiasi variabel `m_H` yang menyimpan nilai *hash* sementara, serta menginisiasi variabel `m_Length` dan `m_TotalLength` yang menyimpan panjang pesan dengan nilai `0`.

```
void SHA1::Init()
{
    // inisiasi variabel m_H
    for (int i = 0; i < 5; i++)
    {
        m_H[i] = m_H0[i];
    }

    m_Length = 0;
    m_TotalLength = 0;
}
```

Fungsi `Update()` yang diimplementasi dibawah ini menerima dua buah parameter masukan, yang pertama adalah data yang akan ditambahkan sebagai pesan yang akan di-*hash*, dan yang kedua adalah panjang data tersebut. Jika data yang dimasukkan sudah mencapai panjang satu block (sesuai yang didefinisikan pada deklarasi kelas `SHA1`), maka block tersebut akan langsung di-`Transform()` dan variabel `m_Block` akan diisi dengan nilai data selanjutnya.

```
void SHA1::Update(unsigned char*
p_Message, unsigned int p_Length)
{
    unsigned int t_RemLength =
SHA1_BLOCK_SIZE - m_Length;
    memcpy(&m_Block[m_Length],
p_Message, t_RemLength);

    if ((m_Length + p_Length) <
SHA1_BLOCK_SIZE)
    {
        m_Length += p_Length;
        return;
    }

    unsigned int t_NewLength =
p_Length - t_RemLength;
    unsigned int t_BlockNum =
t_NewLength / SHA1_BLOCK_SIZE;

    unsigned char* t_ShiftedMsg =
p_Message + t_RemLength;

    Transform(m_Block, 1);
    Transform(t_ShiftedMsg,
t_BlockNum);

    t_RemLength = t_NewLength %
SHA1_BLOCK_SIZE;

    memcpy(m_Block,
&t_ShiftedMsg[t_BlockNum << 6],
t_RemLength);

    m_Length = t_RemLength;
    m_TotalLength += ((t_BlockNum +
1) << 6);
}
```

Fungsi `Final()` menerima satu parameter yang berguna untuk menyimpan hasil keluaran dari fungsi ini yaitu hasil nilai *message digest* akhir. Fungsi `Final()` akan melakukan proses penambahan bit-bit pengganjal pada akhir pesan dan melakukan `Transform()` pada block terakhir untuk mendapatkan *message digest*. Nilai *message digest* tersebut kemudian akan disimpan pada variabel yang diberikan pada parameter fungsi ini.

```
void SHA1::Final(unsigned char*
p_MessageDigest)
{
    unsigned int t_BlockNum = (1 +
((SHA1_BLOCK_SIZE - 9) < (m_Length
% SHA1_BLOCK_SIZE)));

    unsigned int t_LengthBlock =
(m_TotalLength + m_Length) << 3;
    unsigned int t_PMLength =
t_BlockNum << 6;

    memset((m_Block + m_Length), 0,
(t_PMLength - m_Length));
    m_Block[m_Length] = 0x80;
    UNPACK32(t_LengthBlock, m_Block
+ t_PMLength - 4);

    Transform(m_Block, t_BlockNum);

    for (int i = 0; i < 5; i++)
    {
```

```
        UNPACK32(m_H[i],
&p_MessageDigest[i << 2]);
    }
}
```

Fungsi `Transform()` ini merupakan inti dari algoritma SHA-1. Fungsi `Transform()` menerima dua parameter masukan, yaitu *block* pesan yang akan dikomputasi, dan panjang *block* pesan tersebut.

Untuk membangkitkan nilai *message digest*, digunakan metode pertama yang didiskusikan pada RFC-3174 [3]. Pemrosesan melibatkan 80 tahap, dan menggunakan fungsi-fungsi yang telah dijelaskan pada bagian 3.1.1 diatas. Implementasinya adalah sebagai berikut:

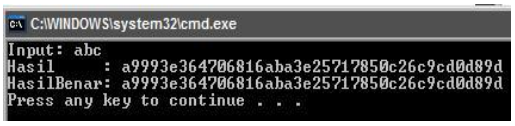
```
void SHA1::Transform(unsigned char*
p_Message, unsigned int p_BlockNum)
{
    unsigned int t_W[80];
    unsigned int t_WV[5];
    unsigned char* t_SubBlock;

    for (unsigned int i = 1; i <=
p_BlockNum; i++)
    {
        t_SubBlock = p_Message +
((i - 1) << 6);

        for (int j = 0; j < 16;
j++)
        {
            PACK32(&t_SubBlock[j <<
2], &t_W[j]);
        }

        for (int j = 16; j < 80;
j++)
        {
            t_W[j] =  ROTL((t_W[j -
3] ^ t_W[j - 8] ^ t_W[j - 14] ^
t_W[j - 16]), 1);
        }

        for (int j = 0; j < 5; j++)
        {
            t_WV[j] = m_H[j];
        }

        for (int j = 0; j < 20;
j++)
        {
            unsigned int t_Temp1 =
ROTL(t_WV[0], 5) + ((t_WV[1] &
t_WV[2]) | ((~t_WV[1]) &
(t_WV[3]))) + t_WV[4] + t_W[j] +
m_K[0];
            t_WV[4] = t_WV[3];
            t_WV[3] = t_WV[2];
            t_WV[2] = ROTL(t_WV[1],
30);
            t_WV[1] = t_WV[0];
            t_WV[0] = t_Temp1;
        }

        for (int j = 20; j <
40; j++)
        {
            unsigned int t_Temp1 =
ROTL(t_WV[0], 5) + (t_WV[1] ^
t_WV[2] ^ t_WV[3]) + t_WV[4] +
t_W[j] + m_K[1];
            t_WV[4] = t_WV[3];
            t_WV[3] = t_WV[2];
            t_WV[2] = ROTL(t_WV[1],
30);
            t_WV[1] = t_WV[0];
            t_WV[0] = t_Temp1;
        }

        for (int j = 40; j <
60; j++)
        {
            unsigned int t_Temp1 =
ROTL(t_WV[0], 5) + ((t_WV[1] &
t_WV[2]) | (t_WV[1] & t_WV[3]) |
(t_WV[2] & t_WV[3])) + t_WV[4] +
t_W[j] + m_K[2];
            t_WV[4] = t_WV[3];
            t_WV[3] = t_WV[2];
            t_WV[2] = ROTL(t_WV[1],
30);
            t_WV[1] = t_WV[0];
            t_WV[0] = t_Temp1;
        }

        for (int j = 60; j <
80; j++)
        {
            unsigned int t_Temp1 =
ROTL(t_WV[0], 5) + (t_WV[1] ^
t_WV[2] ^ t_WV[3]) + t_WV[4] +
t_W[j] + m_K[3];
            t_WV[4] = t_WV[3];
            t_WV[3] = t_WV[2];
            t_WV[2] = ROTL(t_WV[1],
30);
            t_WV[1] = t_WV[0];
            t_WV[0] = t_Temp1;
        }

        for (int j = 0; j < 5; j++)
        {
            m_H[j] += t_WV[j];
        }
    }
}
```

3.1.3. Pengujian

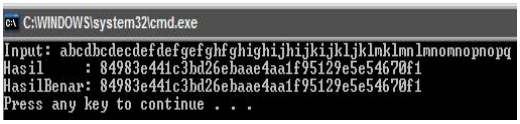
Pada dokumen FIPS PUBS 180-2 [2], dibahas tiga contoh kasus dalam pengujian algoritma SHA, baik SHA-1, maupun varian lainnya. Ketiga contoh kasus tersebut menyatakan input yang diterima oleh algoritma SHA, serta menspesifikasikan output yang seharusnya dikeluarkan. Pertama adalah dengan input string ASCII "`abc`", kedua adalah dengan input string ASCII seperti berikut:

"abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq", dan yang yang terakhir adalah dengan input string ASCII "a" yang berulang sebanyak 1.000.000 (satu juta) kali. Pada pengujian kali ini hanya digunakan dua contoh kasus pertama, sedangkan yang ketiga akan digunakan untuk membandingkan performa ketiga algoritma SHA yang dibahas.

Hasil pengujian implementasi algoritma SHA-1 dapat dilihat pada **Gambar 4** dan **Gambar 5**.



Gambar 4. Pengujian SHA-1 Pertama



Gambar 5. Pengujian SHA-1 Kedua

3.2. SHA-256

Algoritma SHA-256 dapat digunakan untuk menghitung nilai *message digest* dari sebuah pesan, dimana pesan tersebut memiliki panjang maksimum 2^{64} bit. Algoritma ini menggunakan sebuah *message schedule* yang terdiri dari 64 elemen 32-bit *word*, delapan buah variabel 32-bit, dan variabel penyimpan nilai *hash* 8 buah *word* 32-bit. Hasil akhir dari algoritma SHA-256 adalah sebuah *message digest* sepanjang 256-bit.

Preprocessing dilakukan dengan menambahkan bit pengganjal, membagi-bagi pesan dalam block berukuran 512-bit, dan terakhir menginisiasi nilai *hash* awal. Proses penambahan bit pengganjal adalah sama dengan aturan penambahan bit pengganjal pada SHA-1. Proses komputasi SHA-256 dapat dilihat pada bagian implementasi dibawah.

3.2.1. Deskripsi

Dalam proses komputasinya, SHA-256 menggunakan enam fungsi lojik, dimana setiap fungsi beroperasi menggunakan tiga buah *word* 32-bit (x, y, dan z), dan keluarannya berupa sebuah *word* 32-bit. Berikut ini adalah fungsi-fungsi dalam SHA-256 [2]:

$$Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z)$$
$$Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

$$\sum_0^{256}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$$
$$\sum_1^{256}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$$
$$\sigma_0^{256}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$$
$$\sigma_1^{256}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$$

Nilai *hash* awal pada algoritma SHA-256 adalah sebagai berikut:

```
H0[0] = 6a09e667
H0[1] = bb67ae85
H0[2] = 3c6ef372
H0[3] = a54ff53a
H0[4] = 510e527f
H0[5] = 9b05688c
H0[6] = 1f83d9ab
H0[7] = 5be0cd19
```

Dan konstanta dalam SHA-256 adalah sebagai berikut (64 buah):

```
428a2f98 71374491 b5c0fbcf e9b5dba5
3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3
72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc
2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7
c6e00bf3 d5a79147 06ca6351 14292967

27b70a85 2e1b2138 4d2c6dfc 53380d13
650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3
d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08 2748774c 34b0bcb5
391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3
748f82ee 78a5636f 84c87814 8cc70208
90befffa a4506ceb bef9a3f7 c67178f2
```

3.2.2. Implementasi

Dalam implementasinya, kelas `SHA256` didefinisikan sebagai turunan kelas `IHash`. Pada kelas `SHA256` didefinisikan juga dua buah makro ukuran block, yaitu `SHA256_BLOCK_SIZE` dengan `SHA256_DIGEST_SIZE`, serta empat buah fungsi yang digunakan pada proses komputasi SHA-256. Berikut ini adalah deklarasi kelas `SHA256` (file `SHA256.h`):

```
#if !defined(_SHA256_H)
#define _SHA256_H

#include "IHash.h"

#define SHA256_BLOCK_SIZE (512 / 8)
#define SHA256_DIGEST_SIZE (256 /
8)
```

```
#define SHA256_F1(x) (ROTR(x,  2) ^
ROTR(x, 13) ^ ROTR(x, 22))
#define SHA256_F2(x) (ROTR(x,  6) ^
ROTR(x, 11) ^ ROTR(x, 25))
#define SHA256_F3(x) (ROTR(x,  7) ^
ROTR(x, 18) ^ SHFR(x,  3))
#define SHA256_F4(x) (ROTR(x, 17) ^
ROTR(x, 19) ^ SHFR(x, 10))


class SHA256 : public IHash
{
public:
    SHA256();
    virtual ~SHA256();
       void Init();
       void Update(unsigned char*
p_Message, unsigned int p_Length);
       void Final(unsigned char*
p_MessageDigest);
protected:
    void Transform(unsigned char*
p_Message, unsigned int
p_BlockNum);
private:
       unsigned int m_H[8];
       unsigned int m_H0[8];
       unsigned int m_K[64];
       unsigned char m_Length;
};

#endif  //_SHA256_H
```

Kemudian, seluruh fungsi yang dideklarasikan pada `SHA256.h` akan diimplementasi pada `SHA256.cpp`. Berikut ini adalah implementasi konstruktor kelas `SHA256` yang menginisiasi nilai-nilai variabel `m_H0` dan `m_K[0]` sampai `m_K[63]` sesuai dengan nilai *hash* awal dan konstanta pada algoritma SHA-256, serta mengalokasikan variabel `m_Block`.

```
#include "SHA256.h"

SHA256::SHA256()
{
    m_Block = new unsigned char[2 *
SHA256_BLOCK_SIZE];

    // inisiasi H
    m_H0[0] = 0x6a09e667;m_H0[1] =
0xbb67ae85;m_H0[2] =
0x3c6ef372;m_H0[3] = 0xa54ff53a;
    m_H0[4] = 0x510e527f;m_H0[5] =
0x9b05688c;m_H0[6] =
0x1f83d9ab;m_H0[7] = 0x5be0cd19;

    // inisiasi K
    m_K[0] = 0x428a2f98;m_K[1] =
0x71374491;m_K[2] =
0xb5c0fbcf;m_K[3] = 0xe9b5dba5;
    m_K[4] = 0x3956c25b;m_K[5] =
0x59f111f1;m_K[6] =
0x923f82a4;m_K[7] = 0xab1c5ed5;
    m_K[8] = 0xd807aa98;m_K[9] =
0x12835b01;m_K[10] =
0x243185be;m_K[11] = 0x550c7dc3;
    m_K[12] = 0x72be5d74;m_K[13] =
0x80deb1fe;m_K[14] =
0x9bdc06a7;m_K[15] = 0xc19bf174;
    m_K[16] = 0xe49b69c1;m_K[17] =
0xefbe4786;m_K[18] =
0x0fc19dc6;m_K[19] = 0x240ca1cc;

    m_K[20] = 0x2de92c6f;m_K[21] =
0x4a7484aa;m_K[22] =
0x5cb0a9dc;m_K[23] = 0x76f988da;
    m_K[24] = 0x983e5152;m_K[25] =
0xa831c66d;m_K[26] =
0xb00327c8;m_K[27] = 0xbf597fc7;
    m_K[28] = 0xc6e00bf3;m_K[29] =
0xd5a79147;m_K[30] =
0x06ca6351;m_K[31] = 0x14292967;
    m_K[32] = 0x27b70a85;m_K[33] =
0x2e1b2138;m_K[34] =
0x4d2c6dfc;m_K[35] = 0x53380d13;
    m_K[36] = 0x650a7354;m_K[37] =
0x766a0abb;m_K[38] =
0x81c2c92e;m_K[39] = 0x92722c85;

    m_K[40] = 0xa2bfe8a1;m_K[41] =
0xa81a664b;m_K[42] =
0xc24b8b70;m_K[43] = 0xc76c51a3;
    m_K[44] = 0xd192e819;m_K[45] =
0xd6990624;m_K[46] =
0xf40e3585;m_K[47] = 0x106aa070;
    m_K[48] = 0x19a4c116;m_K[49] =
0x1e376c08;m_K[50] =
0x2748774c;m_K[51] = 0x34b0bcb5;
    m_K[52] = 0x391c0cb3;m_K[53] =
0x4ed8aa4a;m_K[54] =
0x5b9cca4f;m_K[55] = 0x682e6ff3;
    m_K[56] = 0x748f82ee;m_K[57] =
0x78a5636f;m_K[58] =
0x84c87814;m_K[59] = 0x8cc70208;

    m_K[60] = 0x90befffa;m_K[61] =
0xa4506ceb;m_K[62] =
0xbef9a3f7;m_K[63] = 0xc67178f2;
}
```

Kemudian, dibawah ini adalah implementasi destruktor yang mendealokasi memori yang telah dialokasi sebelumnya.

```
SHA256::~SHA256()
{
    delete m_Block;
}
```

Fungsi `Init()` dibawah ini melakukan inisiasi variabel `m_H` yang menyimpan nilai *hash* sementara, serta menginisiasi variabel `m_Length` dan `m_TotalLength` yang menyimpan panjang pesan dengan nilai `0`.

```
void SHA256::Init()
{
    // inisiasi variabel H
    for (int i = 0; i < 8; i++)
    {
        m_H[i] = m_H0[i];
    }

    m_Length = 0;
    m_TotalLength = 0;
}
```

Fungsi `Update()` menerima dua buah parameter masukan, yang pertama adalah data yang akan ditambahkan sebagai pesan yang akan di-*hash*, dan yang kedua adalah panjang data tersebut. Jika data yang dimasukkan sudah mencapai panjang satu block, maka block tersebut akan langsung dimasukkan ke fungsi `Transform()` dan variabel `m_Block` akan diisi dengan nilai data selanjutnya. Implementasinya adalah sebagai berikut:

```
void SHA256::Update(unsigned char*
p_Message, unsigned int p_Length)
{
    unsigned int t_RemLength =
SHA256_BLOCK_SIZE - m_Length;
    memcpy(&m_Block[m_Length],
p_Message, t_RemLength);

    if ((m_Length + p_Length) <
SHA256_BLOCK_SIZE)
    {
        m_Length += p_Length;
        return;
    }

    unsigned int t_NewLength =
p_Length - t_RemLength;
    unsigned int t_BlockNum =
t_NewLength / SHA256_BLOCK_SIZE;

    unsigned char* t_ShiftedMsg =
p_Message + t_RemLength;

    Transform(m_Block, 1);
    Transform(t_ShiftedMsg,
t_BlockNum);

    t_RemLength = t_NewLength %
SHA256_BLOCK_SIZE;

    memcpy(m_Block,
&t_ShiftedMsg[t_BlockNum << 6],
t_RemLength);

    m_Length = t_RemLength;
    m_TotalLength += ((t_BlockNum +
1) << 6);
}
```

Fungsi `Final()` menerima satu parameter yang berguna untuk menyimpan hasil keluaran dari fungsi ini yaitu hasil nilai *message digest* akhir. Fungsi `Final()` akan melakukan proses penambahan bit-bit pengganjal pada akhir pesan dan melakukan `Transform()` pada block terakhir untuk mendapatkan *message digest*. Nilai *message digest* tersebut kemudian akan disimpan pada variabel yang diberikan pada parameter fungsi ini.

```
void SHA256::Final(unsigned char*
p_MessageDigest)
{
    unsigned int t_BlockNum = (1 +
((SHA256_BLOCK_SIZE - 9) <
(m_Length % SHA256_BLOCK_SIZE)));

    unsigned int t_LengthBlock =
(m_TotalLength + m_Length) << 3;
    unsigned int t_PMLength =
t_BlockNum << 6;

    memset((m_Block + m_Length), 0,
(t_PMLength - m_Length));
    m_Block[m_Length] = 0x80;
    UNPACK32(t_LengthBlock, m_Block
+ t_PMLength - 4);

    Transform(m_Block, t_BlockNum);

    for (int i = 0; i < 8; i++)
    {
        UNPACK32(m_H[i],
&p_MessageDigest[i << 2]);
    }
}
```

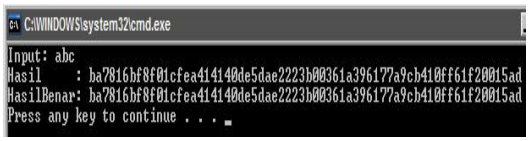
Fungsi `Transform()` pada SHA-256 menerima dua parameter masukan, yaitu *block* pesan yang akan dikomputasi, dan panjang *block* pesan tersebut. Untuk membangkitkan nilai *message digest*, dilakukan proses komputasi yang melibatkan 64 putaran untuk tiap block. Implementasinya adalah sebagai berikut:

```
void SHA256::Transform(unsigned
char* p_Message, unsigned int
p_BlockNum)
{
    unsigned int t_W[64];
    unsigned int t_WV[8];
    unsigned char* t_SubBlock;

    for (unsigned int i = 1; i <=
p_BlockNum; i++)
    {
        t_SubBlock = p_Message +
((i - 1) << 6);

        for (int j = 0; j < 16;
j++)
        {
```

```
            PACK32(&t_SubBlock[j << 2], &t_W[j]);
        }

        for (int j = 16; j < 64; j++)
        {
            t_W[j] = SHA256_F4(t_W[j - 2]) + t_W[j - 7] + SHA256_F3(t_W[j - 15]) + t_W[j - 16];
        }

        for (int j = 0; j < 8; j++)
        {
            t_WV[j] = m_H[j];
        }

        for (int j = 0; j < 64; j++)
        {
            unsigned int t_Temp1 = t_WV[7] + SHA256_F2(t_WV[4]) + CH(t_WV[4], t_WV[5], t_WV[6]) + m_K[j] + t_W[j];
            unsigned int t_Temp2 = SHA256_F1(t_WV[0]) + MAJ(t_WV[0], t_WV[1], t_WV[2]);
            t_WV[7] = t_WV[6];
            t_WV[6] = t_WV[5];
            t_WV[5] = t_WV[4];
            t_WV[4] = t_WV[3] + t_Temp1;
            t_WV[3] = t_WV[2];
            t_WV[2] = t_WV[1];
            t_WV[1] = t_WV[0];
            t_WV[0] = t_Temp1 + t_Temp2;
        }

        for (int j = 0; j < 8; j++)
        {
            m_H[j] += t_WV[j];
        }
    }
}
```

3.2.3. Pengujian

Pengujian algoritma SHA-256 dilakukan dengan contoh kasus yang sama dengan SHA-1, yaitu dengan string "`abc`" dan string "`abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq`". Hasilnya dapat dilihat pada **Gambar 6** dan **Gambar 7**.



Gambar 6. Pengujian SHA-256 Pertama

Gambar 7. Pengujian SHA-256 Kedua

3.3. SHA-512

Algoritma SHA-512 dapat digunakan untuk menghitung nilai *message digest* dari sebuah pesan, dimana pesan tersebut memiliki panjang maksimum 2^{128} bit. Algoritma ini menggunakan sebuah *message schedule* yang terdiri dari 80 elemen 64-bit *word*, delapan buah variabel 64-bit, dan variabel penyimpan nilai *hash* 8 buah *word* 64-bit. Hasil akhir dari algoritma SHA-512 adalah sebuah *message digest* sepanjang 512-bit.

Preprocessing dilakukan dengan menambahkan bit pengganjal, membagi-bagi pesan dalam block berukuran 1024-bit, dan terakhir menginisiasi nilai *hash* awal. Misalkan panjang pesan *M* adalah *l* bit. Proses penambahan bit pengganjal adalah pertama tambahkan bit "1" pada akhir pesan, diikuti dengan bit "0" sejumlah *k*, dimana *k* adalah solusi dari persamaan:

$$l + 1 + k \equiv 896 \bmod 1024$$

Kemudian tambahkan 128-bit block yang menyatakan panjang pesan semula (*l*) dalam representasi biner. Proses komputasi SHA-512 dapat dilihat pada bagian implementasi dibawah.

3.3.1. Deskripsi

Dalam proses komputasinya, SHA-512 menggunakan enam fungsi lojik, dimana setiap fungsi beroperasi menggunakan tiga buah *word* 64-bit (x, y, dan z), dan keluarannya berupa sebuah *word* 64-bit. Fungsi-fungsi tersebut serupa dengan fungsi yang digunakan pada SHA-256, kecuali menggunakan jumlah rotasi pada ROTR dan SHR yang berbeda. Berikut ini adalah fungsi-fungsi dalam SHA-512 [2]:

$$Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$$
$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

$$\sum_0^{512}(x) = ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x)$$
$$\sum_1^{512}(x) = ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x)$$
$$\sigma_0^{512}(x) = ROTR^{1}(x) \oplus ROTR^{8}(x) \oplus SHR^{7}(x)$$
$$\sigma_1^{512}(x) = ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^{6}(x)$$

Nilai *hash* awal pada algoritma SHA-256 adalah sebagai berikut:

```
H0[0] = 6a09e667f3bcc908
H0[1] = bb67ae8584caa73b
H0[2] = 3c6ef372fe94f82b
H0[3] = a54ff53a5f1d36f1
H0[4] = 510e527fade682d1
H0[5] = 9b05688c2b3e6c1f
H0[6] = 1f83d9abfb41bd6b
H0[7] = 5be0cd19137e2179
```

Dan konstanta dalam SHA-512 adalah sebagai berikut (80 buah):

```
428a2f98d728ae22  7137449123ef65cd
b5c0fbcfec4d3b2f  e9b5dba58189dbbc
3956c25bf348b538  59f111f1b605d019
923f82a4af194f9b  ab1c5ed5da6d8118
d807aa98a3030242  12835b0145706fbe
243185be4ee4b28c  550c7dc3d5ffb4e2
72be5d74f27b896f  80deb1fe3b1696b1
9bdc06a725c71235  c19bf174cf692694
e49b69c19ef14ad2  efbe4786384f25e3
0fc19dc68b8cd5b5  240ca1cc77ac9c65
2de92c6f592b0275  4a7484aa6ea6e483
5cb0a9dcbd41fbd4  76f988da831153b5
983e5152ee66dfab  a831c66d2db43210
b00327c898fb213f  bf597fc7beef0ee4
c6e00bf33da88fc2  d5a79147930aa725
06ca6351e003826f  142929670a0e6e70
27b70a8546d22ffc  2e1b21385c26c926
4d2c6dfc5ac42aed  53380d139d95b3df
650a73548baf63de  766a0abb3c77b2a8
81c2c92e47edaee6  92722c851482353b

a2bfe8a14cf10364  a81a664bbc423001
c24b8b70d0f89791  c76c51a30654be30
d192e819d6ef5218  d69906245565a910
f40e35855771202a  106aa07032bbd1b8
19a4c116b8d2d0c8  1e376c085141ab53
2748774cdf8eeb99  34b0bcb5e19b48a8
391c0cb3c5c95a63  4ed8aa4ae3418acb
5b9cca4f7763e373  682e6ff3d6b2b8a3
748f82ee5defb2fc  78a5636f43172f60
84c87814a1f0ab72  8cc702081a6439ec
90befffa23631e28  a4506cebde82bde9
bef9a3f7b2c67915  c67178f2e372532b
ca273eceea26619c  d186b8c721c0c207
eada7dd6cde0eb1e  f57d4f7fee6ed178
06f067aa72176fba  0a637dc5a2c898a6
113f9804bef90dae  1b710b35131c471b
28db77f523047d84  32caab7b40c72493
3c9ebe0a15c9bebc  431d67c49c100d4c
4cc5d4becb3e42b6  597f299cfc657e2a
5fcb6fab3ad6faec  6c44198c4a475817
```

3.3.2. Implementasi

Dalam implementasinya, kelas `SHA512` didefinisikan sebagai turunan kelas `IHash`. Sama seperti `SHA256`, pada kelas `SHA512` didefinisikan juga dua buah makro ukuran block, yaitu `SHA256_BLOCK_SIZE` dengan `SHA256_DIGEST_SIZE`, serta empat buah fungsi yang digunakan pada proses komputasi SHA-512. Perbedaan utama dengan kelas `SHA256` adalah tipe data yang digunakan, dimana pada `SHA512` digunakan `unsigned long long` yang berukuran 64-bit. Berikut ini adalah deklarasi kelas `SHA512` (file `SHA512.h`).

```
#if !defined(_SHA512_H)
#define _SHA512_H

#include "IHash.h"

#define SHA512_BLOCK_SIZE (1024 /
8)
#define SHA512_DIGEST_SIZE (512 /
8)
#define SHA512_F1(x) (ROTR(x, 28) ^
ROTR(x, 34) ^ ROTR(x, 39))
#define SHA512_F2(x) (ROTR(x, 14) ^
ROTR(x, 18) ^ ROTR(x, 41))
#define SHA512_F3(x) (ROTR(x,  1) ^
ROTR(x,  8) ^ SHFR(x,  7))
#define SHA512_F4(x) (ROTR(x, 19) ^
ROTR(x, 61) ^ SHFR(x,  6))


class SHA512 : public IHash
{
public:
    SHA512();
    virtual ~SHA512();
       void Init();
       void Update(unsigned char*
p_Message, unsigned int p_Length);
       void Final(unsigned char*
p_MessageDigest);
protected:
    void Transform(unsigned char*
p_Message, unsigned int
p_BlockNum);
private:
       unsigned long long m_H[8];
       unsigned long long m_H0[8];
       unsigned long long m_K[80];
       unsigned int m_Length;
};

#endif  //_SHA512_H
```

Kemudian, seluruh fungsi yang dideklarasikan pada `SHA512.h` akan diimplementasi pada `SHA512.cpp`. Berikut ini adalah implementasi konstruktor kelas `SHA512` yang menginisiasi nilai-nilai variabel `m_H0` dan `m_K[0]` sampai `m_K[80]` sesuai dengan nilai *hash* awal dan konstanta pada algoritma SHA-512, serta mengalokasikan variabel `m_Block`.

```
#include "SHA512.h"

SHA512::SHA512()
{
    m_Block = new unsigned char[2 *
SHA512_BLOCK_SIZE];
```

```
    // inisiasi H
m_H0[0] = 0x6a09e667f3bcc908ULL;
m_H0[1] = 0xbb67ae8584caa73bULL;
m_H0[2] = 0x3c6ef372fe94f82bULL;
m_H0[3] = 0xa54ff53a5f1d36f1ULL;
m_H0[4] = 0x510e527fade682d1ULL;
m_H0[5] = 0x9b05688c2b3e6c1fULL;
m_H0[6] = 0x1f83d9abfb41bd6bULL;
m_H0[7] = 0x5be0cd19137e2179ULL;

    // inisiasi K
m_K[0] = 0x428a2f98d728ae22ULL;
m_K[1] = 0x7137449123ef65cdULL;
m_K[2] = 0xb5c0fbcfec4d3b2fULL;
m_K[3] = 0xe9b5dba58189dbbcULL;
m_K[4] = 0x3956c25bf348b538ULL;
m_K[5] = 0x59f111f1b605d019ULL;
m_K[6] = 0x923f82a4af194f9bULL;
m_K[7] = 0xab1c5ed5da6d8118ULL;
m_K[8] = 0xd807aa98a3030242ULL;
m_K[9] = 0x12835b0145706fbeULL;
m_K[10] = 0x243185be4ee4b28cULL;
m_K[11] = 0x550c7dc3d5ffb4e2ULL;
m_K[12] = 0x72be5d74f27b896fULL;
m_K[13] = 0x80deb1fe3b1696b1ULL;
m_K[14] = 0x9bdc06a725c71235ULL;
m_K[15] = 0xc19bf174cf692694ULL;
m_K[16] = 0xe49b69c19ef14ad2ULL;
m_K[17] = 0xefbe4786384f25e3ULL;
m_K[18] = 0x0fc19dc68b8cd5b5ULL;
m_K[19] = 0x240ca1cc77ac9c65ULL;

m_K[20] = 0x2de92c6f592b0275ULL;
m_K[21] = 0x4a7484aa6ea6e483ULL;
m_K[22] = 0x5cb0a9dcbd41fbd4ULL;
m_K[23] = 0x76f988da831153b5ULL;
m_K[24] = 0x983e5152ee66dfabULL;
m_K[25] = 0xa831c66d2db43210ULL;
m_K[26] = 0xb00327c898fb213fULL;
m_K[27] = 0xbf597fc7beef0ee4ULL;
m_K[28] = 0xc6e00bf33da88fc2ULL;
m_K[29] = 0xd5a79147930aa725ULL;
m_K[30] = 0x06ca6351e003826fULL;
m_K[31] = 0x142929670a0e6e70ULL;
m_K[32] = 0x27b70a8546d22ffcULL;
m_K[33] = 0x2e1b21385c26c926ULL;
m_K[34] = 0x4d2c6dfc5ac42aedULL;
m_K[35] = 0x53380d139d95b3dfULL;
m_K[36] = 0x650a73548baf63deULL;
m_K[37] = 0x766a0abb3c77b2a8ULL;
m_K[38] = 0x81c2c92e47edaee6ULL;
m_K[39] = 0x92722c851482353bULL;

m_K[40] = 0xa2bfe8a14cf10364ULL;
m_K[41] = 0xa81a664bbc423001ULL;
m_K[42] = 0xc24b8b70d0f89791ULL;
m_K[43] = 0xc76c51a30654be30ULL;
m_K[44] = 0xd192e819d6ef5218ULL;
m_K[45] = 0xd69906245565a910ULL;
m_K[46] = 0xf40e35855771202aULL;
m_K[47] = 0x106aa07032bbd1b8ULL;
m_K[48] = 0x19a4c116b8d2d0c8ULL;
m_K[49] = 0x1e376c085141ab53ULL;
m_K[50] = 0x2748774cdf8eeb99ULL;
m_K[51] = 0x34b0bcb5e19b48a8ULL;
m_K[52] = 0x391c0cb3c5c95a63ULL;
m_K[53] = 0x4ed8aa4ae3418acbULL;
m_K[54] = 0x5b9cca4f7763e373ULL;
m_K[55] = 0x682e6ff3d6b2b8a3ULL;
m_K[56] = 0x748f82ee5defb2fcULL;
m_K[57] = 0x78a5636f43172f60ULL;
m_K[58] = 0x84c87814a1f0ab72ULL;
m_K[59] = 0x8cc702081a6439ecULL;

m_K[60] = 0x90befffa23631e28ULL;
m_K[61] = 0xa4506cebde82bde9ULL;
m_K[62] = 0xbef9a3f7b2c67915ULL;
m_K[63] = 0xc67178f2e372532bULL;
m_K[64] = 0xca273eceea26619cULL;
m_K[65] = 0xd186b8c721c0c207ULL;
m_K[66] = 0xeada7dd6cde0eb1eULL;
m_K[67] = 0xf57d4f7fee6ed178ULL;
m_K[68] = 0x06f067aa72176fbaULL;
m_K[69] = 0x0a637dc5a2c898a6ULL;
m_K[70] = 0x113f9804bef90daeULL;
m_K[71] = 0x1b710b35131c471bULL;
m_K[72] = 0x28db77f523047d84ULL;
m_K[73] = 0x32caab7b40c72493ULL;
m_K[74] = 0x3c9ebe0a15c9bebcULL;
m_K[75] = 0x431d67c49c100d4cULL;
m_K[76] = 0x4cc5d4becb3e42b6ULL;
m_K[77] = 0x597f299cfc657e2aULL;
m_K[78] = 0x5fcb6fab3ad6faecULL;
m_K[79] = 0x6c44198c4a475817ULL;
}
```

Kemudian, dibawah ini adalah implementasi destruktor yang mendealokasi memori yang telah dialokasi sebelumnya.

```
SHA512::~SHA512()
{
    delete m_Block;
}
```

Fungsi `Init()` dibawah ini melakukan inisiasi variabel `m_H` yang menyimpan nilai *hash* sementara, serta menginisiasi variabel `m_Length` dan `m_TotalLength` yang menyimpan panjang pesan dengan nilai `0`.

```
void SHA512::Init()
{
    // inisiasi variabel H
    for (int i = 0; i < 8; i++)
    {
        m_H[i] = m_H0[i];
    }

    m_Length = 0;
    m_TotalLength = 0;
}
```

Fungsi `Update()` menerima dua buah parameter masukan, yang pertama adalah data yang akan ditambahkan sebagai pesan yang akan di-*hash*, dan yang kedua adalah panjang data tersebut. Jika data yang dimasukkan sudah mencapai panjang satu block, maka block tersebut akan langsung dimasukkan ke fungsi

`Transform()` dan variabel `m_Block` akan diisi dengan nilai data selanjutnya. Implementasinya adalah sebagai berikut:

```
void SHA512::Update(unsigned char*
p_Message, unsigned int p_Length)
{
    unsigned int t_RemLength =
SHA512_BLOCK_SIZE - m_Length;
    memcpy(&m_Block[m_Length],
p_Message, t_RemLength);

    if ((m_Length + p_Length) <
SHA512_BLOCK_SIZE)
    {
        m_Length += p_Length;
        return;
    }

    unsigned int t_NewLength =
p_Length - t_RemLength;
    unsigned int t_BlockNum =
t_NewLength / SHA512_BLOCK_SIZE;

    unsigned char* t_ShiftedMsg =
p_Message + t_RemLength;

    Transform(m_Block, 1);
    Transform(t_ShiftedMsg,
t_BlockNum);

    t_RemLength = t_NewLength %
SHA512_BLOCK_SIZE;

    memcpy(m_Block,
&t_ShiftedMsg[t_BlockNum << 7],
t_RemLength);

    m_Length = t_RemLength;
    m_TotalLength += ((t_BlockNum +
1) << 7);
}
```

Fungsi `Final()` menerima satu parameter yang berguna untuk menyimpan hasil keluaran dari fungsi ini yaitu hasil nilai *message digest* akhir. Fungsi `Final()` akan melakukan proses penambahan bit-bit pengganjal pada akhir pesan dan melakukan `Transform()` pada block terakhir untuk mendapatkan *message digest*. Nilai *message digest* tersebut kemudian akan disimpan pada variabel yang diberikan pada parameter fungsi ini.

```
void SHA512::Final(unsigned char*
p_MessageDigest)
{
    unsigned int t_BlockNum = (1 +
((SHA512_BLOCK_SIZE - 17) <
(m_Length % SHA512_BLOCK_SIZE)));

    unsigned int t_LengthBlock =
(m_TotalLength + m_Length) << 3;
    unsigned int t_PMLength =
t_BlockNum << 7;

    memset((m_Block + m_Length), 0,
(t_PMLength - m_Length));
    m_Block[m_Length] = 0x80;
    UNPACK32(t_LengthBlock, m_Block
+ t_PMLength - 4);

    Transform(m_Block, t_BlockNum);

    for (int i = 0; i < 8; i++)
    {
        UNPACK64(m_H[i],
&p_MessageDigest[i << 3]);
    }
}
```

Fungsi `Transform()` pada SHA-512 menerima dua parameter masukan, yaitu *block* pesan yang akan dikomputasi, dan panjang *block* pesan tersebut. Untuk membangkitkan nilai *message digest*, dilakukan proses komputasi yang melibatkan 80 putaran untuk tiap block. Implementasinya adalah sebagai berikut:

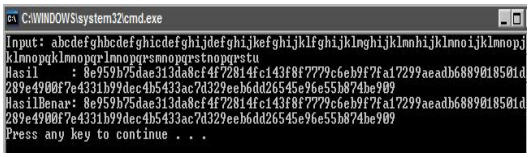
```
void SHA512::Transform(unsigned
char* p_Message, unsigned int
p_BlockNum)
{
    unsigned long long t_W[80];
    unsigned long long t_WV[8];
    unsigned char* t_SubBlock;

    for (unsigned int i = 1; i <=
p_BlockNum; i++)
    {
        t_SubBlock = p_Message +
((i - 1) << 7);

        for (int j = 0; j < 16;
j++)
        {
            PACK64(&t_SubBlock[j <<
3], &t_W[j]);
        }

        for (int j = 16; j < 80;
j++)
        {
            t_W[j] =
SHA512_F4(t_W[j - 2]) + t_W[j - 7]
+ SHA512_F3(t_W[j - 15]) + t_W[j -
16];
        }

        for (int j = 0; j < 8; j++)
        {
            t_WV[j] = m_H[j];
        }

        for (int j = 0; j < 80;
j++)
        {
```

```
            unsigned long long
t_Temp1 = t_WV[7] +
SHA512_F2(t_WV[4]) + CH(t_WV[4],
t_WV[5], t_WV[6]) + m_K[j] +
t_W[j];
            unsigned long long
t_Temp2 = SHA512_F1(t_WV[0]) +
MAJ(t_WV[0], t_WV[1], t_WV[2]);
            t_WV[7] = t_WV[6];
            t_WV[6] = t_WV[5];
            t_WV[5] = t_WV[4];
            t_WV[4] = t_WV[3] +
t_Temp1;
            t_WV[3] = t_WV[2];
            t_WV[2] = t_WV[1];
            t_WV[1] = t_WV[0];
            t_WV[0] = t_Temp1 +
t_Temp2;
        }

        for (int j = 0; j < 8; j++)
        {
            m_H[j] += t_WV[j];
        }
    }
}
```

3.3.3. Pengujian

Pengujian SHA-512 sedikit berbeda dengan yang lain. *Test vector* pertama adalah tetap yaitu string ASCII "`abc`", sedangkan yang kedua berbeda, yaitu pesan string ASCII sepanjang 896-bit:
"`abcdefghbcdefghicdefghijdefghijkef ghijklfghijklmghijklmnhijklmnoijklm nopjklmnopqklmnopqrlmnopqrsmnopqrst nopqrstu`"
Hasil pengujian kelas `SHA512` dengan contoh kasus diatas dapat dilihat pada **Gambar 8** dan **Gambar 9**.

Gambar 8. Pengujian SHA-512 Pertama



Gambar 9. Pengujian SHA-512 Kedua

4. Perbandingan

Perbandingan dilakukan dengan mengeksekusi ketiga implementasi algoritma diatas dan kemudian menghitung waktu eksekusinya masing-masing. Untuk menghitung waktu eksekusi, maka diperlukan suatu kelas lain yang dapat menghitung waktu eksekusi program seakurat mungkin dengan resolusi setinggi mungkin. Oleh karena itu dibuatlah kelas `StopWatch`, yang berfungsi seperti *stop watch* dalam dunia nyata, yang dapat menghitung waktu eksekusi program dengan resolusi tinggi (*high resolution timer*).

4.1. Stopwatch

Sebagai solusi dari kebutuhan akan perhitungan waktu resolusi tinggi, kelas `StopWatch` dibuat dan dapat digunakan untuk menghitung waktu yang dibutuhkan dalam mengeksekusi satu atau beberapa *statement* program tertentu. Kelas ini diimplementasi dalam bahasa C++ dan akan digunakan dalam program. Agar dapat mencapai perhitungan waktu dengan resolusi tinggi, maka fungsi-fungsi dalam kelas `StopWatch` diimplementasi dengan bahasa *assembly* yang ditempelkan dalam bahasa C++ (*embedded assembly*). Bahasa *assembly* merupakan bahasa yang sangat tergantung pada mesin. Sintaks bahasa assembly yang digunakan pada implementasi ini merupakan sintaks untuk prosesor Intel dan untuk *compiler* Microsoft Visual C++.

4.1.1. Implementasi Stopwatch

Pada definisi kelas `StopWatch` (`StopWatch.h`), fungsi yang perlu dideklarasikan adalah `Start()`, `Reset()`, dan `GetCurrentValue()`. Fungsi `Start()` berguna untuk memulai penghitungan waktu, dan waktu yang telah berlalu sejak fungsi `Start()` dipanggil bisa didapatkan dengan fungsi `GetCurrentValue()`. Variabel `m_Time` digunakan untuk menyimpan waktu yang telah dilalui, dan disimpan sebagai variabel 64-bit. Fungsi `Reset()` hanya mengeset variabel `m_Time` tersebut menjadi 0.

Pada pengujian, digunakan mesin dengan prosesor berkecepatan 1,86 GHz, maka makro `MACHINE_SPEED` harus diset sesuai dengan kecepatan tersebut.

```
#if !defined(_STOPWATCH_H)
#define _STOPWATCH_H

#define MHZ 1000000 //satu jt hertz
#define MACHINE_SPEED 1866*MHZ
//kecepatan CPU!!

class StopWatch
{
private:
    __int64 m_Time;
```

```
public:
    StopWatch();
    ~StopWatch();
    void Start();
    void Reset();
    double GetCurrentValue();
};

#endif  //_STOPWATCH_H
```

Implementasinya kelas `StopWatch` (`StopWatch.cpp`), dengan fungsi `Start()` dan fungsi `GetCurrentValue()` diimplementasi dengan *embedded assembly*, adalah sebagai berikut:

```
#include "StopWatch.h"

StopWatch::StopWatch()
{ }

StopWatch::~StopWatch()
{ }

void StopWatch::Start()
{
    // cpu instructions
    #define rdtsc __asm __emit 0Fh
__asm __emit 031h
    #define cpuid __asm __emit 0Fh
__asm __emit 0A2h

    __int64 ts = 0; //working var

    __asm push EAX
    __asm push EDX
    //cpuid  //other info
    rdtsc    //read time stamp
register

    __asm mov dword ptr ts, EAX
//low bits
    __asm and EDX, 07fffffffh //63
bit int, sign removed
    __asm mov dword ptr ts+4,EDX
//high bits

    __asm pop EDX
    __asm pop EAX

    #undef rdtsc
    #undef cpuid
    m_Time = ts;
}

void StopWatch::Reset()
{
       m_Time = 0;
}

double StopWatch::GetCurrentValue()
{
       #define rdtsc __asm __emit
0Fh __asm __emit 031h
       #define cpuid __asm __emit
0Fh __asm __emit 0A2h

       __int64 ts = 0;

       __asm push EAX
       __asm push EDX
       //cpuid  //other info
       rdtsc    //read timestamp

       __asm mov dword ptr ts, EAX
//low bits
       __asm and EDX, 07fffffffh
//63 bit int, sign removed
       __asm mov dword ptr ts+4,EDX
//high bits

       __asm pop EDX
       __asm pop EAX

       #undef rdtsc
       #undef cpuid
       ts -= m_Time;
       return
(double)(ts)/(double)(MACHINE_SPEED
);
}
```

4.2. Pengujian

Pengujian yang diharapkan adalah membandingkan performa ketiga algoritma yang telah diimplementasi, dengan seakurat mungkin. Pengujian dilakukan pada komputer dengan spesifikasi sebagai berikut:

- Sistem Operasi: Microsoft Windows XP Professional Edition Service Pack 2
- *Processor*: Intel Pentium M 1.86 GHz
- Arsitektur CPU: 32-bit
- *Memory*: 1 GB
- *Compiler*: Microsoft Visual C++ 8.0

Gambar 10, **Gambar 11**, dan **Gambar 12** adalah keluaran yang dihasilkan dari percobaan ketiga implementasi algoritma SHA diatas, dengan konfigurasi *compiler* sebagai berikut:

- Optimasi *Compiler*: *disabled*
- *Debug mode*

Gambar 10. Pengujian Performa SHA1

Gambar 11. Pengujian Performa SHA-256

Gambar 12. Pengujian Performa SHA-512

Dari gambar diatas, dapat dilihat bahwa dengan algoritma SHA-1, jika input yang diberikan adalah berupa 1.000.000 (satu juta) karakter ASCII "a", maka waktu yang dibutuhkan dari mulai menginstansiasi object SHA1 sampai menyimpan hasilnya pada variabel keluaran adalah selama 0,0348499 detik. Sedangkan untuk algoritma SHA-256 adalah 0,0648242 detik dan algoritma SHA-512 adalah 0,155484 detik.

Ketiga hasil diatas adalah hasil dari satu kali percobaan. Pada percobaan lainnya, hasil yang didapat tidak terlalu jauh berbeda, dimana perbedaan yang terjadi hanya kurang lebih sebesar 0,01 detik.

Untuk membandingkan performa ketiga implementasi algoritma SHA diatas, maka pengujian selanjutnya yang dilakukan adalah dengan melakukan optimasi pada pengaturan *compiler*, serta menambahkan panjang pesan yang diberikan sebagai masukan menjadi 100.000.000 karakter 'a'. Optimasi yang dilakukan adalah:

1. *Release Mode*
2. *Compiler Optimization*:
   a. *Maximize speed* (/O2)
   b. *Inline function expansion* (/Ob2)
   c. *Favor fast code* (/Ot)
   d. *Omit Frame Pointer* (/Oy)
   e. *Eliminate Unreferenced Data* (/OPT:REF)
   f. *Optimize for x86* (/MACHINE:X86)

Hasilnya dapat dilihat pada masing-masing **Gambar 1** dan **Gambar 2**. Rangkuman hasil seluruh pengujian yang dilakukan dapat dilihat pada **Tabel 2**.

Alg.	Perc. 1	Perc. 2	Perc. 3
SHA-1	0,0348499	0,0102459	1,09406
SHA-256	0,0648242	0,0205226	2,04955
SHA-512	0,155484	0,0436857	4,05868

Tabel 2. Perbandingan Ketiga Algoritma dalam Ketiga Pengujian

Dari hasil pengujian, didapat bahwa performa algoritma yang terbaik adalah SHA-1, dilanjutkan dengan SHA-256 di posisi kedua dan SHA-512 di posisi terakhir. Hal ini terjadi karena beberapa hal, yaitu:

1. Jumlah *loop* dalam algoritma SHA-256 lebih kecil daripada *loop* dalam algoritma SHA-512.
2. Algoritma SHA-512 menggunakan tipe data 64-bit dalam operasinya, sedangkan eksekusi program dilakukan pada mesin 32-bit.
3. SHA-1 menggunakan panjang variabel yang lebih pendek dibanding dua algoritma lainnya, serta operasi yang lebih sedikit.

Optimasi sebenarnya dapat dilakukan lebih lanjut, dengan melakukan beberapa hal. Pertama misalnya dengan membuka *loop* (*unroll loop*) untuk meningkatkan performa secara keseluruhan. Kedua adalah kompilasi dan eksekusi pada mesin berarsitektur 64-bit. Optimasi ini akan sangat terasa terutama pada algoritma SHA-512 yang menggunakan panjang *word* 64-bit. Brian Gladman [4] dalam situsnya menampilkan hasil pengujian varian algoritma SHA dengan CPU AMD64, yang bisa dilihat pada **Tabel 3** (dalam satuan siklus CPU (CPU *cycle*) per byte data).

Algoritma	*10.000 byte*	*100.000 byte*
SHA1	9,4	9.7
SHA256	20.4	20.4
SHA512	13.5	13.4

Tabel 3. Performa algoritma SHA pada AMD64

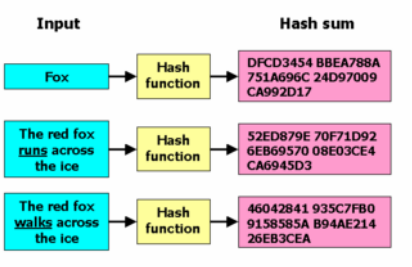
Dapat dilihat pada tabel diatas bahwa algoritma SHA-512 memiliki performa yang jauh lebih baik daripada algoritma SHA-256 jika dikompilasi terhadap arsitektur 64-bit dan dijalankan pada mesin 64-bit.

4.3. Aspek Keamanan

Ketiga algoritma SHA tersebut sebenarnya sudah tergolong aman [3]. Algoritma SHA dikatakan aman karena tidak mungkin secara komputasional untuk menemukan pesan yang berpasangan dengan sebuah *message digest*

tertentu. Selain itu, tidak mungkin juga secara komputasional untuk menghasilkan dua pesan berbeda yang menghasilkan *message digest* yang sama. Sedikit perubahan pada pesan kemungkinan besar akan menghasilkan *message digest* yang sangat berbeda.

Keamanan algoritma SHA dibuktikan dengan *avalanche effect*, yaitu jika terjadi perubahan sedikit pada pesan, maka *message digest* yang dihasilkan dapat berbeda jauh. Untuk lebih jelasnya, *avalanche effect* dapat dilihat pada **Gambar 13**.



Gambar 13. *Avalanche Effect*

Pada gambar diatas, mengubah satu kata bagian pesan dari *runs* menjadi *walks* dapat menghasilkan nilai *message digest* yang jauh berbeda.

Akan tetapi, tetap saja tidak ada keamanan yang sempurna. Algoritma SHA masih dapat diserang. Pada Februari 2005, algoritma SHA-1 dapat diserang dan ditemukan *collision* dengan kurang dari 2^{69} operasi. *Collision*, yaitu dua data pesan yang berbeda yang menghasilkan *message digest* yang sama berhasil ditemukan. Oleh karena itu SHA-2 (SHA-256 dan SHA-512) dirancang untuk menutupi kekurangan yang dimiliki oleh SHA-1, dengan menambah jumlah putaran (*loop*) dalam algoritmanya dan meningkatkan panjang *message digest* yang dihasilkannya. Sampai saat ini belum ditemukan *collision* pada algoritma SHA-256 maupun SHA-512.

Jadi, dapat dikatakan bahwa SHA-256 dan SHA-512 lebih aman dibandingkan SHA-1, karena tingkat kompleksitasnya yang lebih tinggi serta *message digest* yang dihasilkannya lebih panjang.

5. Kesimpulan

Dari pembahasan diatas, maka dapat ditarik beberapa kesimpulan, yaitu:

1. Fungsi *hash* satu arah mengubah pesan yang panjang menjadi nilai *hash*, dan hampir tidak mungkin untuk mengkonstruksi pesan kembali dari nilai *hash* tersebut.
2. Fungsi SHA-1, SHA-256, dan SHA-512 memiliki karakteristik dan implementasi yang hampir mirip. Akan tetapi, ketiganya menggunakan konstanta yang berbeda, panjang variabel berbeda, fungsi yang berbeda, jumlah *loop* berbeda, serta menghasilkan nilai *message digest* yang panjangnya berbeda.
3. Waktu eksekusi algoritma SHA-1 lebih cepat dibanding kedua algoritma lainnya. Akan tetapi, optimasi dapat dilakukan dengan mengeksekusi program pada mesin berarsitektur 64-bit, sehingga waktu eksekusi SHA-512 dapat lebih cepat. Hal ini disebabkan SHA-512 menggunakan variabel dengan panjang 64-bit.
4. *Avalanche effect* dapat terjadi hanya dengan mengubah sedikit bagian dari pesan. Akan tetapi, algoritma SHA-1 masih dapat ditemui *collision*, sehingga dianggap masih kurang aman. SHA-256 dan SHA-512 dianggap lebih aman dibanding SHA-1 karena tingkat kompleksitas yang lebih tinggi dan *message digest* yang dihasilkan lebih panjang.

DAFTAR PUSTAKA

[1] Munir, Rinaldi. (2004). *Bahan Kuliah IF5054 Kriptografi*. Departemen Teknik Informatika, Institut Teknologi Bandung.

[2] Federal Information Processing Standards Publication. (2002). *Federal Information Processing Standards (FIPS) Publication 180-2 : Secure Hash Standard*. National Institute of Standards and Technology, USA.

[3] Eastlake, D., Jones, P. (2001). *Request For Comment (RFC)-3174 : US Secure Hash Algorithm 1 (SHA1)*. The Internet Society. http://tools.ietf.org/html/rfc3174, diakses 28 Desember 2006.

[4] Gladman, Brian. (2006). *SHA1, SHA2, HMAC and Key Derivation in C*. http://fp.gladman.plus.com/cryptography_technology/sha/index.htm. Diakses 28 Desember 2006.