Z-Hash

Fungsi *Hash* dengan Zaslavskii *Map*

Roland Hartanto / 13515107
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Infomatika
Institut Teknologi Bandung
Jln. Ganesha 10 Bandung 40132, Indonesia
rolandhartanto@gmail.com

***Abstract*—Aplikasi fungsi *hash* dalam dunia komputer sangat banyak. Seiring dengan majunya perkembangan teknologi, fungsi *hash* juga terus dikembangkan untuk menghindari terbentuknya lubang keamanan. Perancangan fungsi *hash* harus sangat baik khususnya dalam bidang keamanannya. Kemanan fungsi *hash* ditunjukkan oleh kemungkinan terjadinya kolisi. Semakin banyak kolisi yang terjadi, suatu fungsi *hash* semakin tidak aman secara kriptografis. Z-*Hash* adalah fungsi *hash* yang diimplementasikan dengan menggunakan Zaslavskii *Map*, salah satu jenis *chaotic map*. Banyaknya iterasi yang digunakan dalam Z-*Hash* membuat fungsi ini cukup aman dalam membentuk *hash*.**

Kata kunci—hash; chaotic map; Zaslavskii Map; Kolisi; kriptografi

I. PENDAHULUAN

Salah satu jenis algoritma kriptografi yang sampai saat ini masih dikembangkan adalah fungsi *hash*. Berbeda dengan algoritma kriptografi kunci publik, fungsi *hash* lebih mudah untuk dirancang. Fungsi *hash* merupakan fungsi yang dapat mengubah suatu pesan menjadi *string* dengan panjang tertentu yang sudah ditetapkan berdasarkan algoritmanya. Misalnya, MD5 memiliki panjang *hash* 16 byte dan SHA-1 memiliki panjang *hash* 20 byte.

Fungsi *hash* saat ini sangat banyak pemakaiannya. Penggunaan fungsi *hash* pada umumnya adalah untuk menjaga integritas data, menghemat waktu pengiriman pesan, dan menormalkan panjang data yang beraneka ragam. Integritas data sangat berkaitan dengan perubahan bit pesan sehingga untuk perubahan 1 bit pesan saja seharusnya *hash* yang dihasilkan sudah berbeda. Karena proses pembentukan *hash* yang tidak lama disebabkan oleh faktor penyeragaman ukuran data, penggunaan fungsi *hash* ini sangat menghemat waktu pengiriman pesan.

Tantangan dalam perancangan fungsi *hash* adalah pembuatan fungsi *hash* yang aman. Fungsi *hash* yang aman adalah fungsi *hash* yang sangat kecil kemungkinannya untuk terjadi kolisi. Kolisi merupakan suatu kejadian saat *string hash* yang dihasilkan oleh sebuah fungsi *hash* untuk dua plainteks yang berbeda adalah sama. Adanya kolisi menunjukkan bahwa fungsi *hash* tersebut tidak aman secara kriptografis. Tantangan dalam mengurangi kemungkinan kolisi membuat fungsi *hash* masih dikembangkan sampai sekarang.

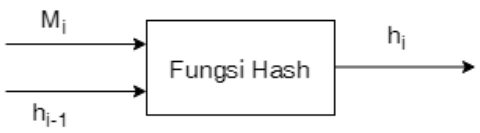
Z-*Hash* adalah fungsi *hash* yang dirancang dengan menggunakan Zaslavskii *map* dalam operasinya. Zaslavskii *map* atau yang biasa disebut Zaslavsky *map* merupakan salah satu fungsi yang bersifat *chaotic*. Sifat *chaotic* ini dimanfaatkan dalam pembentukan *message digest* dari sebuah pesan plainteks. Sifat ini diharapkan memberi perilaku yang acak terhadap *hash* yang dihasilkan sehingga dapat mengurangi kemungkinan terjadinya kolisi.

II. DASAR TEORI

A. Fungsi Hash

Fungsi *hash* adalah fungsi digunakan untuk mengubahan masukan berupa *string* yang panjangnya sembarang menjadi *string* yang memiliki panjang tetap. Fungsi *hash* memiliki sifat satu arah. Hal ini berarti setelah pesan diubah menjadi *message digest*, pesan tidak dapat diperoleh kembali dari *hash* tersebut. Selain itu, untuk setiap pesan yang diberikan, tidak mungkin dicari pesan y yang tidak sama dengan pesan x sedemikian rupa sehingga H(y) = H(x). Oleh sebab itu, tidak mungkin juga apabila suatu pasangan pesan x dan y memiliki *hash* yang sama yang dengan kata lain *hash* tidak mengalami kolisi.

Masukan fungsi *hash* adalah berupa blok-blok pesan yang merupakan potongan-potongan dari satu kesatuan pesan dan keluaran dari hasil *hash* blok pesan sebelumnya. Berikut ini adalah skema fungsi *hash*.



Gambar 1. Skema fungsi *hash*

M_i merupakan blok pesan ke-i, h_{i-1} adalah hasil keluaran fungsi *hash* sebelumnya dan h_i adalah hasil keluaran fungsi *hash*. Apabila blok pesan tersebut adalah blok pertama, maka hasil keluaran fungsi *hash* sebelumnya digantikan dengan suatu vektor awal yang diberi nilai tetap.

Aplikasi fungsi *hash* sangat banyak. Tiga aplikasi utama fungsi *hash* adalah menjaga integritas data, menghemat waktu pengiriman pesan, dan menormalkan panjang data yang beraneka ragam. Terjaganya integritas data ditujukkan dari pengubahan 1 bit pesan kemudian melakukan *hash* pada pesan tersebut. Apabila *hash* berubah, berarti pesan tersebut isinya sudah berubah. Dengan demikian, pengguna dapat yakin dengan kebenaran isi pesan yang dimilikinya dengan menggunakan fungsi *hash*. Penghematan waktu kirim pesan sangat terasa apabila pesan yang dikirimkan ukurannya sangat besar sehingga untuk melakukan enkripsi memerlukan waktu yang cukup lama. Dengan adanya *hash*, verifikasi keaslian suatu pesan dapat lebih cepat karena hanya cukup membandingkan kesamaan nilai *hash* pesan yang dikirm oleh pengirim dengan hasil *hash* pesan yang dikirim di tempat penerima. Apabila hasil *hash* sama, berarti pesan tersebut adalah pesan yang sama dengan pesan aslinya. Penyeragaman ukuran panjang data yang beraneka ragam terpakai dalam penyimpanan data dalam suatu basis data seperti password yang memiliki panjang yang beragam.

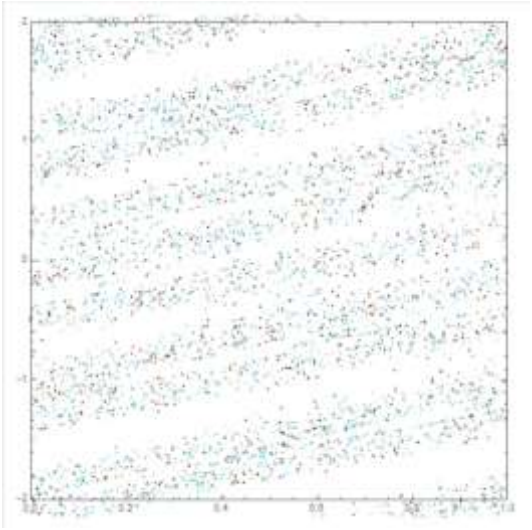
B. *Zaslavskii Map*

Zaslavskii *Map* adalah sebuah pemetaan standar untuk generalisasi disipatif. Pemetaan ini menunjukkan perilaku yang bersifat *chaotic*. Pemetaan ini dapat ditulis dalam persamaan sebagai berikut.

$$x_{n+1} = [x_n + \nu(1 + \mu y_n) + \epsilon\nu\mu\cos(2\pi x_n)] \pmod 1$$
$$y_{n+1} = e^{-r}(y_n + \epsilon\cos(2\pi x_n))$$

$$\mu = \frac{1 - e^{-r}}{r}$$

Pada persamaan ini, terdapat operasi modulo, yaitu mod 1. Hal ini dimaksudkan agar nilai x hanya berada di antara 0 dan 1 saja. Persamaan ini memiliki tiga variabel konstanta, yaitu, v, r, dan ε. Apabila nilai v = 0.9, r = 0.526, ε = 5, dan jumlah iterasi yang dilakukan adalah 10000 kali, pola yang dihasilkan adalah sebagai berikut.



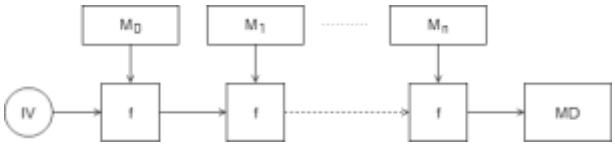
Gambar 2. Zaslavskii *Map* dengan parameter v = 0.9, r = 0.526, ε= 5

(Sumber: Wolfram CDF Player, The Zaslavskii *Map*)

III. RANCANGAN ALGORITMA

A. *Algoritma Keseluruhan*

Algoritma Z-*Hash* secara keseluruhan sederhana karena mengadopsi konstruksi *hash* Merkle-Damgard. Berikut ini adalah skemanya.



Gambar 3. Konstruksi Merkle-Damgard

Pesan terlebih dahulu dilakukan padding apabila pesan berukuran bukan kelipatan 32. Hal ini dilakukan karena ukuran Z-*Hash* adalah 32 byte. Padding dilakukan dengan menambahkan sejumlah karakter '\0' pada byte pesan hingga ukuran byte pesan menjadi kelipatan 32. Pesan yang telah dilakukan padding dibagi-bagi menjadi blok-blok berukuran 32 byte. Pada awal, vektor inisial (IV) bersama M0 dimasukkan ke dalam fungsi f untuk diolah lebih lanjut. Hasil keluaran fungsi f ini digunakan kembali untuk iterasi selanjutnya sebagai masukan bersama blok pesan berikutnya. Setelah seluruh blok pesan selesai diolah, hasil keluaran fungsi f terakhir sudah merupakan *hash* yang dihasilkan.

B. *Fungsi f*

Gambar 4. Fungsi f

Fungsi f adalah fungsi kompresi yang terdiri dari Zaslavskii *map*. Pada awal fungsi f, dihitung terlebih dahulu jumlah nilai

ASCII dari 16 karakter plainteks pertama dan 16 karakter plainteks terakhir yang disimpan ke dalam masing-masing sebuah variabel. Jumlah nilai ASCII ini kemudian menentukan jumlah iterasi yang diperlukan pada Zaslavskii *map*. Iterasi pada Zaslavskii *map* dibuat dengan tujuan untuk memberikan efek yang lebih acak dan kacau terhadap *hash*.

Pada Zaslavskii *map* yang digunakan, nilai inisial x adalah 0.15711592491662785 dan y adalah 0.2790506621876845. Dari nilai-nilai inisial tersebut, dibentuk vektor inisial berupa *string* dengan nilai 157115924916627852790506621876845. Selanjutnya, parameter yang digunakan dalam Zaslavskii *map* adalah v = 0.9, r = 0.526, dan ε = 5. Implementasi Zaslavskii *map* adalah dengan menggunakan rumus yang telah dibahas pada bab sebelumnya.

Proses pada fungsi f diawali dengan penentuan nilai x dan y setelah diiterasi dengan menggunakan Zaslavskii *map*. Jumlah iterasi Zaslavskii *map* mengikuti variabel jumlah nilai ASCII yang telah diisi sebelumnya. Karena ada dua nilai iterasi, maka perlu pemilihan salah satu jumlah iterasi. Apabila pada saat itu adalah iterasi ke-n dengan n adalah bilangan yang habis dibagi 3, maka jumlah iterasi Zaslavski *map* yang digunakan pada iterasi tersebut adalah jumlah yang berasal dari 16 karakter plainteks terakhir. Apabila n adalah angka selain kelipatan 3, maka jumlah iterasi yang dipilih adalah yang berasal dari 16 karakter plainteks pertama.

Selanjutnya, dilakukan iterasi Zaslavski *Map*. Hasil perolehan nilai x dan y dari Zaslavskii *map* selanjutnya diubah dari double menjadi byte sehingga ukuran nilai x dan y dalam byte adalah 8 byte. Karena untuk menyesuaikan dengan panjang blok pesan, yaitu 32 byte, dilakukan padding byte x dan y dengan x dan y sendiri sebanyak 3 kali sehingga total panjang masing-masing x dan y dalam byte saat ini adalah 32 byte. Berikut ini adalah x dan y setelah dilakukan pemanjangan.

```
... (iterasi Zaslavskii Map)
x_byte = bytearray(struct.pack("d", x))
y_byte = bytearray(struct.pack("d", y))


x_byte_concat = x_byte + x_byte + x_byte + x_byte
y_byte_concat = y_byte + y_byte + y_byte + y_byte
```

Kemudian setelah itu dilakukan XOR tahap pertama yaitu XOR antara byte pada nilai x hasil pemanjangan tadi, blok pesan, dan hasil keluaran *hash* pada iterasi sebelumnya. Hasil XOR tersebut disimpan ke dalam suatu variabel sementara. Setelah itu, dilakukan XOR tahap kedua, yaitu XOR antara byte pada nilai y hasil pemanjangan sebelumnya dan hasil XOR tahap pertama yang disimpan pada variabel sementara sebelumnya.

Langkah-langkah tadi dilakukan berulang-ulang untuk setiap blok pesan sampai blok pesan terakhir. Setelah iterasi selesai, dihasilkan *hash* yang diinginkan.

IV. EKSPERIMEN

Fungsi Z-Hash diimplementasikan dalam bahasa Python versi 3.6. Lingkungan operasi yang digunakan dalam eksperimen ini adalah sistem operasi Windows 10, processor Intel® Core™ i7-7700HQ Gen Octacore @2.80 GHz, dan RAM dengan ukuran 16 GB.

A. Contoh Masukan dan Keluaran

Berikut ini adalah contoh masukan berupa data teks dan keluaran dari Z-*Hash* yang diimplementasikan.

Tabel 1. Contoh masukan dan keluaran data berupa teks

Masukan	**Keluaran**
Hello world!	1162c3f5f6d25b5d2e67cdbf9ef522385d00af9198f225365e00a9909ff12d35
The quick brown fox jumps over the lazy dog	35f4bc32693fef0e03bafa0b11409f676b91815612459b397dde90011a41de3e
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.	48dff24939c06de34f95eb1021817a866b8ea44d2bda6aea6185e70a59a55c83

Berikut ini contoh masukan berupa *file* selain teks biasa dan keluarannya.

Masukan 1:

Lenna.png

Keluaran:

```
8309e7291572f8fb0a43436a605f3caf4a53c4c49b
ebec5eebab2806d94ae085
```

Masukan 2: (pengubahan beberapa byte pada gambar)

Lenna2c.png

Keluaran:

```
ef64aab782867f7a662e0ef4f6ab172e263e895a0d1
fc7df87c665984fbecb04
```

Masukan 3: (jenis file berformat DOC)

File: IEEE Paper Template.doc

Potongan file:

```
…

Authors Name/s per 2nd Affiliation (Author)
line 1 (of Affiliation): dept. name of organization
line 2-name of organization, acronyms acceptable
line 3-City, Country
line 4-e-mail address if desired

Abstract—This electronic document is a “live” template and already defines the components of your paper [title, text, heads, etc.] in its style sheet. *CRITICAL: Do Not Use Symbols, Special Characters, or Math in Paper Title or Abstract. (Abstract)
Keywords—component; formatting; style; styling; insert (key words)

…
```

Keluaran:

```
0fc594bb700baf6bd18dc40489b0fb0221e70ae19e2
b364b36e7406ea48968be
```

B. Pengujian dan Analisis

Pengujian dan analisis yang dilakukan antara lain sensitivitas terhadap perubahan kecil pada pesan, kecepatan eksekusi program, dan ketahanan terhadap kolisi. Berikut ini adalah pengujian dan analisis yang dilakukan.

1. Sensitivitas terhadap perubahan kecil pada pesan

   Pada bagian ini, pengujian dilakukan dengan mengubah byte *file* yang akan dijadikan *hash*. Pada data yang berupa teks, pengujian dilakukan dengan mengubah minimal salah satu karakter dalam teks.

   Berikut ini adalah pengujian untuk data berupa teks. Teks-teks di bawah ini merupakan kasus uji yang digunakan dalam eksperimen ini. Kasus uji berupa pengubahan salah satu karakter di dalam teks. Tetapi, pengubahan karakter dilakukan seminimal mungkin seperti mengubah huruf yang letaknya dalam tabel ASCII bersebelahan. Oleh sebab itu, mensubstitusi suatu karakter dengan karakter lain yang berada di sebelahnya pada tabel ASCII kurang lebih sama dengan mengubah sejumlah satu bit pesan aslinya.

   TXT0 - Teks asli:

   ```
   Sensitivitas terhadap perubahan kecil pada pesan
   ```

   TXT1 - Modifikasi 1 (S di awal kalimat menjadi T):

   ```
   Tensitivitas terhadap perubahan kecil pada pesan
   ```

   TXT2 - Modifikasi 2 (S di awal kalimat dihilangkan):

ensitivitas terhadap perubahan kecil pada pesan

TXT3 - Modifikasi 3 (Penambahan tanda titik (.) di akhir):

Sensitivitas terhadap perubahan kecil pada pesan.

Tabel 2. Hasil pengujian data berupa teks

ID Masukan	Keluaran
TXT0	3689af7c327c5ea50690e56275284da462efc60f46083ed37bfbc60e594a2895
TXT1	4509cb41560abdda7210815f115eaedb166fa232227eddac0f7ba2333d3ccbea
TXT2	9c869dcc64b488408cc6828b30ae8d2ff9e8e6b111ddf33beeeee0a857c0be21
TXT3	58f8123c891517a568e15822ce4104a4229e7b4ffd6177d3158a7b4ee2236195

Berikut ini pengujian untuk data berupa *file*.

BIN0 - *File* asli:

Lenna.png (*file* yang sama dengan contoh masukan dan keluaran pada sub bab sebelumnya)

BIN1 – Modifikasi 1:

Lenna2.png (pengubahan byte gambar terakhir dari ‘,’ menjadi ‘.’)

BIN2 – Modifikasi 2:

Lenna3.png (penghapusan byte gambar terakhir)

BIN3 – Modifikasi 3:

Lenna4.png (pengubahan 1 bit pada byte terakhir dari ‘0x2C’ menjadi ‘0x2D’)

Tabel 3. Hasil pengujian data berupa *file*

ID Masukan	Keluaran
BIN0	b0958ca74e0f9f8439df28e43b225bd079cfaf4ac0968b21d8374388823787fa
BIN1	ef64aab783867f7a662e0ef4f6ab172e263e895a0d1fc7df87c665984fbecb04
BIN2	4cd59ae9908e5e7bc59f3eaae5a3182f858fb9041e17c8de247755c65cb6c405
BIN3	72eb674a0ce72804fba1c30979ca4350bbb144a7827e93a11a49a865c0df9f7a

Apabila dilihat dari hasil pengujian yang telah dilakukan, perbedaan *hash* yang dihasilkan teks asli dengan seluruh modifikasi sangat jauh. Hampir seluruh byte dalam *hash* yang dihasilkan berbeda. Hal yang sama juga terjadi untuk data yang berupa *file* gambar.

2. Kecepatan eksekusi program

Bagian ini menguji kecepatan eksekusi algoritma untuk ukuran *file* tertentu bila dibandingkan dengan algoritma fungsi *hash* lain yaitu MD5, SHA1, SHA-256, dan SHA-512.

Tabel 4. Waktu eksekusi algoritma dalam satuan detik

Ukuran *File* (KB)	Z-*Hash*	MD5	SHA 1	SHA-256	SHA-512
1	0.07009	0	0	0	0
2	0.14125	0	0	0	0
4	0.29399	0	0	0	0
16	1.24924	0	0	0	0
64	4.72717	0	0	0	0
128	11.3429	0	0	0	0
256	18.3295	0	0	0	0
512	44.0643	0	0	0	0

Dari hasil pengujian waktu eksekusi, terlihat bahwa fungsi Z-*Hash* membutuhkan waktu yang sangat lama untuk membentuk *hash* bila dibandingkan dengan algoritma *hash* lainnya seperti MD5, SHA1, SHA-256, dan SHA-512. Apabila diamati dari algoritmanya, fungsi *hash* Z-*Hash* memiliki banyak iterasi khususnya pada saat melakukan iterasi Zaslavskii *map*. Hal ini menjadi salah satu faktor kemungkinan lambatnya

algoritma ini. Selain itu, apabila dilihat dari formula yang digunakan pada Zaslavskii *map*, terdapat operasi trigonometri yaitu cos. Selain itu juga terdapat operasi perpangkatan bilangan e (bilangan Euler). Operasi ini lebih mahal dari segi operasinya bila dibandingkan dengan operasi matematika lainnya.

3. Ketahanan terhadap serangan brute force

   Serangan brute force dilakukan untuk mengetahui teks asli dari *hash* yang diberikan (preimage attack). Tipe serangan ini memiliki kompleksitas waktu 2^n dengan n adalah jumlah bit dari *hash* yang dihasilkan. Pada Z-*Hash*, ukuran blok *hash* yang dihasilkan adalah 32 byte atau 256 bit. Apabila penyerang melakukan serangan brute force, kompleksitas penyerangan tersebut adalah 2^{256} atau apabila dalam satu detik komputer dapat beroperasi sejumlah 10^8 kali, waktu yang dibutuhkan untuk melakukan serangan brute force adalah 1157920892373161954235709850086879078532699846656405640394575840079131,29639936 detik. Angka ini menunjukkan bahwa serangan brute force untuk algoritma Z-*Hash* sangat sulit untuk dilakukan dan waktu yang dibutuhkan tidak sebanding dengan informasi yang ingin didapatkan.

4. Ketahanan terhadap kolisi

   Kolisi terjadi apabila terdapat dua buah pesan yang berbeda namun menghasilkan *hash* yang sama. Salah satu cara untuk menguji ketahanan terhadap kolisi adalah dengan menggunakan birthday attack. Dengan birthday attack, kolisi pada fungsi *hash* dapat ditemukan pada $2^{n/2}$ dengan 2^n adalah kompleksitas waktu serangan brute force. Apabila serangan ini dilakukan, maka dibutuhkan sejumlah 340282366920938463463374607431768211456 operasi yang dilakukan oleh komputer. Hal ini berarti juga apabila dalam satu detik, komputer dapat melakukan operasi sejumlah 10^8 kali, maka dibutuhkan waktu 1079028307080601418897 05 tahun untuk melakukan serangan ini.

   Selain dari birthday attack, ketahanan kolisi juga dapat diuji dengan menghitung rata-rata dari jumlah selisih nilai ASCII setiap karakter *hash* yang dihasilkan oleh teks asli dan teks yang diubah salah satu bit di dalamnya secara random. Berikut ini adalah rumus yang digunakan.

$$d = \sum_{i=1}^{n/8} |dec(m_i) - dec(m_i')|$$

   Pada rumus di atas, n adalah jumlah bit pada *string hash* yang dihasilkan. Keluaran fungsi dec(m_i) adalah nilai ASCII dari karakter ke-i dari *hash* yang dihasilkan dari pesan asli sebelum dimodifikasi bitnya. Keluaran fungsi dec(m_i') adalah nilai ASCII dari karakter ke-i dari *hash* yang dihasilkan dari pesan yang telah dimodifikasi 1 bit di dalamnya secara random pada suatu iterasi. Setelah seluruh iterasi selesai, diperoleh d yang merupakan jumlah dari selisih nilai ASCII *hash* pesan sebelum dan sesudah dimodifikasi.

   Pesan yang digunakan dalam pengujian ini adalah sebagai berikut.

   Teks asli:

```
Sensitivitas terhadap perubahan kecil pada pesan
```

   Pada eksperimen kali ini, jumlah iterasi yang dilakukan untuk menghitung jumlah selisih nilai ASCII tersebut adalah sebanyak 5000 kali. Berikut ini adalah hasil yang diperoleh dari pengujian tersebut.

   Tabel 5. Rata-rata selisih jumlah ASCII *hash* pesan asli dan pesan setelah dimodifikasi

Algoritma	**Rata-rata**	**Rata-rata/karakter**
Z-*Hash*	2706.9168	84.5912
MD5	408.7344	25.5459
SHA1	465.7874	23.2893
SHA256	881.073	27.5335
SHA512	1758.5212	27.4769

   Tabel 6. Nilai maksimum dan minimum selisih nilai karakter ASCII *hash* pesan asli dan pesan setelah dimodifikasi

Algoritma	**Maksimum selisih**	**Minimum selisih**
Z-*Hash*	4164	1125
MD5	694	31
SHA1	819	75
SHA256	1285	383
SHA512	2254	923

   Dari tabel tersebut, terlihat bahwa rata-rata selisih nilai ASCII *hash* pesan asli dengan pesan yang telah dimodifikasi untuk setiap karakternya pada fungsi Z-*Hash* memiliki nilai yang cukup tinggi, yaitu, 84.5912. Angka ini merupakan angka yang dekat dengan nilai ideal yaitu 85.43 [13]. Hal ini menunjukkan untuk *string* yang cukup pendek, fungsi Z-*Hash* mampu membuat keacakan yang tinggi sehingga membuat rata-rata selisih nilai ASCII per karakter juga cukup tinggi.

   Selain itu dari tabel 6, terlihat nilai maksimum dan minimum selisih karakter ASCII *hash* pesan asli dengan *hash* pesan yang telah dimodifikasi untuk fungsi Z-*Hash* sangat tinggi. Hal ini juga menunjukkan

keacakan yang tinggi dari nilai *hash* yang dihasilkan setelah data dimodifikasi. SHA256 yang memiliki panjang *message digest* yang sama dengan Z-*Hash* memiliki nilai maksimum dan minimum selisih yang lebih rendah daripada Z-*Hash* untuk ukuran pesan yang kecil.

Pengujian juga dilakukan untuk pesan yang lebih panjang sebagai berikut. Pesan yang digunakan berjenis sama yaitu teks dengan jumlah 1312 karakter sehingga setara dengan *file* yang berukuran 1312 byte.

```
Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore
et dolore magna aliqua. Ut enim ad
minim veniam, quis nostrud
exercitation ullamco laboris nisi ut
aliquip ex ea commodo consequat.
Duis aute irure dolor in
reprehenderit in voluptate velit
esse cillum dolore eu fugiat nulla
pariatur. Excepteur sint occaecat
cupidatat non proident, sunt in
culpa qui officia deserunt mollit
anim id est laborum.

... dan seterusnya (karena teks
terlalu panjang, teks tidak
ditampilkan seluruhnya).

Ukuran pesan 1312 byte.
```

Berikut adalah hasil pengukuran rata-rata selisih nilai ASCII *message digest* pesan yang asli dengan pesan yang telah dimodifikasi bit di dalamnya untuk pesan yang lebih panjang tersebut.

Tabel 7. Rata-rata selisih jumlah ASCII *hash* pesan asli dan pesan setelah dimodifikasi

Algoritma	**Rata-rata**	**Rata-rata/karakter**
Z-*Hash*	2560.3184	80.0099
MD5	468.605	29.2878
SHA1	446.378	22.3189
SHA256	808.2278	25.2571
SHA512	1564.5734	24.4464

Tabel 8. Rata-rata selisih jumlah ASCII *hash* pesan asli dan pesan setelah dimodifikasi

Algoritma	**Maksimum selisih**	**Minimum selisih**
Z-*Hash*	3714	1160
MD5	709	54
SHA1	810	96
SHA256	1262	253
SHA512	2182	875

Dari tabel hasil eksperimen di atas, Z-*Hash* terlihat lebih unggul apabila dibandingkan dengan fungsi *hash* yang lainnya. Namun, walaupun begitu, hampir seluruh rata-rata selisih nilai ASCII *hash* pesan asli dengan pesan yang telah dimodifikasi menurun. Hal ini mungkin disebabkan karena teks yang dipakai mengandung banyak karakter yang sama. Dengan kata lain teks uji distribusi karakternya kurang merata sehingga membuat hasilnya sedikit menurun. Tetapi walaupun begitu, apabila memang teks ini memiliki persebaran yang kurang merata dan dibutuhkan lagi data yang lebih besar yang mungkin memiliki distribusi karakter yang lebih merata untuk melakukan pengujian, dari segi ketahanan terhadap kolisi, Z-*Hash* lebih unggul daripada fungsi *hash* yang lainnya. Hal ini ditunjukkan dengan data yang distribusinya karakternya kurang merata, Z-*Hash* mampu menghasilkan *hash* dengan tingkat keacakan yang cukup tinggi.

V. KESIMPULAN DAN SARAN

Berdasarkan pengujian dan analisis yang telah dilakukan, dapat disimpulkan Z-*Hash* tergolong aman secara kriptografis sebagai fungsi *hash*. Namun, Z-*Hash* tidak memenuhi aplikasi fungsi *hash* sebagai sarana untuk menghemat waktu pengiriman pesan. Hal ini ditunjukkan dari waktu eksekusi fungsi yang sangat lama bila dibandingkan dengan fungsi *hash* lainnya seperti MD5, SHA1, SHA-256, dan SHA-512. Lamanya durasi eksekusi fungsi *hash* ini disebabkan karena adanya beberapa operasi matematika yang mahal dari segi kompleksitasnya. Operasi tersebut antara lain cosinus dan perpangkatan bilangan Euler.

Saran untuk penelitian selanjutnya adalah sebaiknya pengujian kolisi dilakukan dengan data yang ukurannya lebih besar lagi untuk memperoleh hasil yang lebih akurat. Selain itu, algoritma ini mungkin dapat dikembangkan lagi dengan mengurangi jumlah iterasi yang dibutuhkan, dengan harapan waktu eksekusi bisa lebih cepat tanpa mengurangi kualitas dari ketahanan kolisi algoritma ini. Selain itu, pengembangan fungsi *hash* juga dapat dilakukan tidak menggunakan Zaslavskii *map*. Fungsi chaos ada banyak jenisnya sehingga dapat dicoba untuk mengetahui fungsi yang paling baik untuk memperoleh *hash* yang bagus kualitasnya baik dari segi ketahanan terhadap kolisi, kinerja dan kecepatan eksekusi, serta sensitivitas *hash* yang dihasilkan terhadap pengubahan sedikit pesan asli.

VI. Ucapan Terima Kasih

Penulis mengucapkan terima kasih kepada Tuhan Yang Maha Esa, karena atas bantuan, rahmat, dan berkat-Nya, makalah ini dapat selesai pada waktunya. Tak lupa juga, penulis ingin menyampaikan terima kasih kepada Bapak Dr. Ir. Rinaldi Munir selaku dosen mata kuliah IF4020 Kriptografi yang telah membagikan ilmunya kepada penulis. Selain itu, penulis juga ingin menyampaikan terima kasih kepada kedua orang tua yang selalu mendukung penulis.

Referensi

[1] R. Munir, *Diktat Kuliah IF5054 Kriptografi*, Departemen Teknik Informatika Institut Teknologi Bandung, 2005.
[2] R. Munir, *Slide Kuliah IF4020 Kriptografi*, Fungsi *Hash*, 2018.
[3] R. Munir, *Slide Kuliah IF4020 Kriptografi*, SHA, 2018.
[4] R. Munir, *Slide Kuliah IF4020 Kriptografi*, Algoritma MD5, 2018.
[5] R. Munir, *Slide Kuliah IF4020 Kriptografi*, SHA-3, 2018.
[6] San-Um, Wimol, & Srichavengsup, Warakom. 2016. A Robust *Hash* Function Using Cross-Coupled *Chaotic Map*s with Absolute-Valued Sinusoidal Nonlinearity. IJACSA, Vol. 7, No. 1.
[7] http://www.wikiwand.com/en/Zaslavskii_*map* diakses pada tanggal 15 April 2018, pukul 15.00.
[8] https://cs.nyu.edu/~dodis/ps/nist.pdf diakses pada tanggal 15 April 2018, pukul 16.00.
[9] https://en.wikipedia.org/wiki/Cryptographic_*hash*_function diakses pada tanggal 17 Mei 2018, pukul 20.00.
[10] https://streamhpc.com/blog/2012-07-16/how-expensive-is-an-operation-on-a-cpu/ diakses pada tanggal 17 Mei 2018, pukul 19.00.
[11] https://en.wikipedia.org/wiki/Birthday_attack diakses pada tanggal 17 Mei 2018, pukul 17.00.
[12] https://en.wikipedia.org/wiki/Preimage_attack diakses pada tanggal 17 Mei 2018, pukul 16.00.
[13] Y. Wang, K. Wong, D. Xiao, "Parallel *hash* function construction based on coupled *map* lattices", Communications in Nonlinear Science and Numerical Simulation, Vol. 16, No. 7, pp. 2810-2821, 2011