Studi dan Implementasi *Hashing* untuk *URL Shortening Service*

Ricardo Pramana Suranta / 13509014[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13509014@itb.ac.id

***Abstrak*—*URL Shortening Service* merupakan layanan yang disediakan oleh berbagai pihak untuk mempersingkat sebuah alamat *URL* yang umumnya cukup panjang, sehingga dapat dibagikan melalui berbagai media, termasuk perangkat *handheld* seperti *smartphone* atau *tablet* serta situs-situs *microblogging* seperti Twitter.com. Pihak penyedia layanan tersebut umumnya tidak memberitahukan cara penyingkatan sebuah alamat URL menjadi sebuah *short URL*, namun, untuk pembangkitan *short URL* sendiri, dapat menggunakan fungsi *hash* tradisiona yang selama ini dikenal. Fungsi *hash* CRC32 menawarkan *short URL* sepanjang 6 karakter dengan ketahanan *collision* yang cukup baik, dan diperbaiki dengan konkatenasi *hash value domain* situs tersebut dengan CRC32, dengan *hash value* dari halaman yang hendak dituju dengan CRC32 atau CRC16, yang masing-masing menghasilkan *short URL* sepanjang 12 dan 9 karakter, dengan *encoding* basis 64.**

Kata Kunci – URL Shortening Service, hash, CRC32, CRC16, collision

I. PENDAHULUAN

Seiring dengan pesatnya perkembangan teknologi dan perangkat genggam (*handheld*) seperti *smartphone* dan *tablet*, masyarakat awam saat ini memiliki akses yang mudah untuk mengakses internet, dimana saja, kapan saja. Oleh karena sebagian besar perangkat genggam tersebut memiliki ruang atau layar tampilan yang terbatas, maraknya *microblogging* seperti Twitter, serta kenyamanan pengguna, alamat *URL* (*Uniform Resource Locator*) yang hendak dibagikan juga "terpaksa" dipersingkat. Untuk mengatasi hal ini, berbagai pihak berusaha untuk menyediakan *URL shortening service*, seperti bit.ly, tinyurl.com, dan goo.gl. Layanan ini cukup marak digunakan oleh berbagai pihak pada saat ini, agar informasi, terutama alamat situs yang hendak dibagikan dapat diakses oleh berbagai orang melalui berbagai media, baik PC, *notebook*, *handheld*, dan sebagainya.

Gambar 1.1*salah satu contoh penggunaan URL Shortening Service dalam layanan microblogging yang disediakan oleh Twitter. Penyedia URL Shortening Service yang digunakan adalah Google (goo.gl)*

Metode yang digunakan oleh penyedia *URL Shortening Service* untuk mempersingkat alamat *URL* sangatlah beragam, dan tidak banyak diantaranya yang mau membagikan metode tersebut kepada umum, terutama penyedia layanan yang berkualitas tinggi. Konsep penyingkatan alamat *URL* tersebut dapat didekati dengan konsep fungsi *hash*, yakni fungsi yang menerima suatu *string* dengan panjang yang sembarang, lalu mengembalikan sebuah *string* unik dengan panjang yang tetap. Oleh karena fungsi *hash* bertujuan untuk mengembalikan *string* yang unik untuk tiap masukan yang diterima, maka fungsi *hash* memiliki potensi yang besar untuk digunakan sebagai metode penyingkatan alamat URL, yang dapat diketahui oleh khayalak ramai, dan dapat diukur kualitasnya secara formal.

Gambar1 2 *Tiga penyedia URL Shortening Service yang umum digunakan, yakni goo.gl, bit.y, serta tinyurl.com. Ketiganya tidak menyebarluaskan metode penyingkatan alamt URL yang mereka gunakan.*

II. DASAR TEORI

2.1Fungsi Hash

Fungsi *hash* adalah fungsi yang menerima masukan *string* yang panjangnya sembarang dan mengkonversinya menjadi *string* keluaran yang panjangnya tetap (*fixed*). Fungsi *hash* memiliki sifat satu arah, sehingga *string* keluaran dari suatu fungsi *hash* (umumnya disebut nilai hash atau *hash value*) tidak dapat dikembalikan lagi menjadi pesan semula. Sifat-sifat fungsi *hash* adalah sebagai berikut (dengan asumsi fungsi *H* adalah fungsi *hash*):

1. Fungsi *H* dapat diterapkan pada blok data (masukan) berukuran berapa saja.
2. Fungsi *H* menghasilkan nilai (*h*) dengan panjang tetap (*fixed length output*).
3. Untuk tiap *h* yang dihasilkan, tidak mungkin dikembalikan nilai *x* sedemikian sehingga *H(x)= h* (*irreversible*).
4. Untuk setiap *x* yang diberikan, tidak mungkin mencari $y \neq x$ sedemikian sehingga *H(y) = H(x)*.
5. Tidak mungkin mencari pasangan *x* dan *y* sedemikian sehingga *H(x) = H(y).*

Namun, dalam pengaplikasiannya di dunia nyata, tidak semua fungsi *hash* yang digunakan dapat memberikan kepastian bahwa tidak terdapat $y \neq x$ yang menghasilkan *H(y) = H(x)* (poin 4 dan 5, atau keunikan *hash value*), contohnya fungsi MD5 (*Message Digest 5*), yang sangat sering digunakan untuk pemeriksaan integritas data (*checksum*) oleh karena *avalanche effect* yang sangat baik, yakni *hash value* yang "sensitif" terhadap nilai awal, namun dapat menghasilkan sebuah *hash value* yang sama untuk dua *string* masukan yang berbeda[6] . Kejadian dimana $y \neq x$ dan *H(x) = H(y)* disebut sebagai *collision*, dan fungsi *hash* yang dapat memenuhi poin 4 dan 5 diatas disebut sebagai *perfect hash function*.

2.2 URL Shortening Service

URL Shortening Service adalah layanan penyingkatan suatu alamat *URL* (*Uniform Resource Locator*), yang mengubah alamat *URL* umumnya cukup panjang menjadi sebuah alamat *URL* yang singkat (*short URL*). Layanan ini umumnya digunakan pada situs-situs *microblogging* yang menyediakan ruang terbatas bagi para penggunanya (mis. Twitter, yang hanya menyediakan 140 karakter), pemberian rujukan pada surat kabar agar mudah untuk diakses oleh para pembaca, dan tidak jarang untuk penyebaran beberapa iklan serta *malware* yang ada di internet, karena pada umumnya alamat URL yang sudah disingkat tidak memberikan informasi apapun tentang *website* yang dirujuk olehnya.

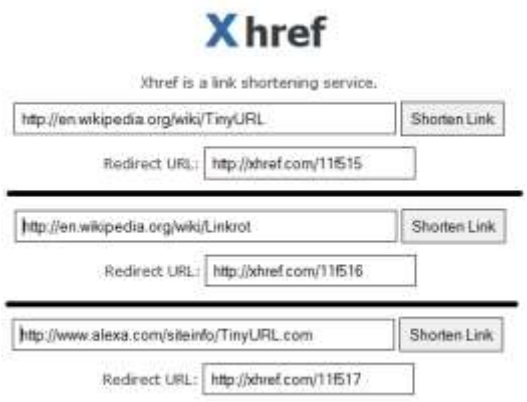
Dalam pengaplikasiannya, layanan ini menggunakan basis data yang terdiri atas tabel-tabel yang terdiri atas sebuah *primary key* dan sebuah kolom yang menyimpan "alamat sesungguhnya" dari *short URL* yang diberikan tersebut. Tidak ada standar baku dalam pembentukan *short URL* ini, namun ide yang umumnya diketahui oleh orang-orang untuk membangkitkan *primary key* tersebut adalah dengan cara melakukan iterasi terhadap *primary key* terakhir yang sudah digunakan, lalu mengembalikan *short URL* yang merupakan *primary key* dari alamat sebenarnya yang telah dikonversi dengan *encoding* basis-*n* (umumnya basis 32 yang mencakup karakter A-Z dan 0-9). Apabila pengguna mengakses *short URL* tersebut, nilai dari *short URL* tersebut akan diubah kembali menjadi *primary key* dari alamat sebenarnya. Metode ini umumnya menggunakan *primary key* dengan tipe *integer*, sehingga *primary key* tersebut sama dengan indeks dari baris dalam tabel yang ada. Hal ini memungkinkan waktu pengaksesan alamat sebenarnya menjadi cukup singkat oleh karena tidak memerlukan pencarian, serta memastikan bahwa *short URL* yang didapatkan oleh pengguna akan mengembalikan alamat yang benar, oleh karena *short URL* yang dikembalikan pastilah unik.

Kelemahan dari metode yang dijelaskan diatas adalah semakin panjangnya *short URL* yang dikembalikan, oleh karena bertambahnya jumlah data yang terdapat didalam tabel utama. Untuk mencegah hal ini, pihak penyedia layanan harus melakukan pemeriksaan duplikasi terhadap alamat URL yang menjadi masukan serta melakukan pemeriksaan alamat yang sudah tidak valid lagi (*dead URL*), sehingga *value* dari *dead URL* tersebut dapat digantikan dengan alamat URL yang baru. Kedua hal ini berpotensi memperlambat proses pembangkitan *short URL*, yang tentunya mengurangi kenyamanan pengguna untuk memakai.

Selain metode yang dijelaskan tersebut, pihak penyedia layanan dapat membentuk cara pembangkitan *primary key* maupun *short URL* yang unik dan mungkin jauh berbeda dari metode tersebut. Salah satu contoh penyedia layanan *URL Shortening Service* yang mengembalikan *short URL* yang unik adalah linkpot.net, yang mengembalikan *short URL* berupa kata dalam bahasa Inggris, sehingga mudah diingat oleh pengguna. Namun, memiliki *short URL* yang mudah diingat pun bukanlah suatu keharusan, oleh karena umumnya pengguna lansgung melakukan *copy-and-paste* atas *short URL* yang dibangkitkan ke tempat dimana pengguna tersebut hendak menyimpan atau membagikannya.

Secara implisit, terdapat beberapa aspek yang harus diperhatikan untuk sebuah *URL shortening service* yang baik :

1. Panjang *short URL* yang cukup pendek,
2. Kepastian bahwa *short URL* mengembalikan alamat yang benar, dan
3. Pembangkitan *short URL* yang cepat.



Gambar 2.1 *xhref.com, salah satu penyedia URL Shortening Service yang menggunakan konversi basis-n terhadap primary key yang urut. Ketiga URL tersebut dimasukkan berurutan, dalam jeda waktu yang cukup singkat.*

III. ANALISIS

Untuk meminimalisir jumlah data yang disimpan dalam basis data utama, "pencarian" atas suatu alamat masukan untuk memastikan tidak terjadinya duplikasi nilai sangatlah penting, sehingga daripada mengambil tempat baru, penyedia layanan dapat mengembalikan *short URL* dari alamat masukan serupa yang sudah ada sebelumnya. Optimalisasi proses pemeriksaan ini pun dapat dilakukan dengan banyak cara, termasuk penggunaan fungsi *hash*, antara lain :

1. Pengunaan *hash value* untuk pembanding kesamaan alamat *URL* masukan dengan alamat *URL* yang ada didalam basis data (namun tidak sebagai *primary key*) untuk mencegah terjadinya duplikasi, dan
2. Penggunaan *hash value* sebuah alamat *URL* sebagai *primary key* dan *short URL*.

Untuk memenuhi ketiga aspek yang dibutuhkan *URL Shortening Service*, dilakukan pengujian serta analisis atas beberapa fungsi *hash* "tradisional" yang umumnya digunakan, yakni pengujian kecepatan pembangkitan *hash value* dan *collision* dari tiap fungsi *hash* tersebut. Fungsi *hash* yang diuji antara lain MD5, SHA1, SHA256, Adler32, dan CRC32, yang disediakan oleh *library* Java.

3.1 Pengujian

Langkah-langkah pengujian yang akan dilakukan terhadap kelima fungsi *hash* tersebut adalah sebagai berikut :

1. Masukkan URL yang dijadikan data uji kedalam sebuah *array* (pengujian ini menggunakan objek *ArrayList* dalam lingkungan Java).
2. Lakukan *hashing* terhadap tiap data uji yang ada dengan masing-masing fungsi. Catat waktu *hashing* tiap fungsi, dan simpan *hash value* dari tiap data uji kedalam objek *ArrayList* penampung *hash value* untuk masing-masing fungsi yang tersedia.
3. Lakukan pemeriksaan duplikasi *hash value* dari *ArrayList* masing-masing fungsi.
4. Cari rata-rata waktu *hashing* dari tiap fungsi.

Data uji yang disediakan untuk pengujian ini diambil dari alamat *webpage* unik yang ada dari situs-situs berikut, tiap situs sejumlah 500 *webpage*:

1. www.codinghorror.com/blog
2. www.omgubuntu.co.uk
3. www.joelonsoftware.org
4. www.kaskus.us
5. www.stackoverflow.com
6. en.wikipedia.org
7. nasional.kompas.com
8. www.youtube.com

alamat dari seluruh situs diatas diambil dari *sitemap* yang dibangkitkan dengan layanan yang disediakan oleh www.xml-sitemaps.com. Untuk memastikan keunikan tiap *webpage* yang menjadi data uji, dilakukan penghapusan terhadap duplikasi data yang ada dalam *ArrayList* penampung data uji.

Hasil dari pengujian diatas adalah sebagai berikut :

Nama Fungsi	Jumlah *collision*	Kecepatan rata-rata (detik)
MD5	0	0.00000998291
SHA-1	0	0.00000933214
SHA-256	0	0.00001317949
Adler32	210	0.00000274436
CRC32	0	0.00000244240

Tabel 3.1 *Hasil pengujian collision dan kecepatan MD5, SHA-1, SHA-256, Adler32, CRC32 terhadap 4000 buah URL, format keluaran basis 16. Proses konversi hasil hash ke basis 16 juga termasuk dalam perhitungan ini.*

Dari pengujian diatas, terlihat bahwa Adler32 menghasilkan 210 buah *collision*, yang berarti memungkinkan *short URL* yang dihasilkan tidaklah unik, dan dapat mengarahkan pengguna ke alamat *URL* yang salah. Oleh karena itu, Adler32 tidak akan dianalisis lebih lanjut.

3.2 Analisis

Berdasarkan hasil pengujian diatas, dapat dilihat bahwa CRC32 memiliki kecepatan rata-rata *hash* yang paling singkat, yakni 0.00244240 milidetik. Hal ini dipengaruhi oleh kompleksitas fungsi *hash*, dan fungsi CRC32 memiliki fungsi *hash* yang paling sederhana. *Pseudo-code* dar fungsi MD5, SHA-1, SHA-256, dan CRC32 disertakan di lampiran.

Selain kecepatan, keempat fungsi *hash* tersebut memberikan hasil sangat baik dalam uji *collision*, yakni tidak mengalami *collision* sama sekali. Secara teoritis, untuk mencegah terjadinya *collision*, sebuah fungsi *hash* harus memiliki jumlah kemungkinan *hash value* yang cukup untuk menampung berbagai kemungkinan masukan. Hal ini disebut prinsip *pigeonhole* yang ada untuk mencegah *birthday paradox*, dimana kemungkinan *collision* dari *hash value* menjadi sekitar 50% saat masukan fungsi mencapai akar dari jumlah *hash value* yang mungkin. Jumlah kemungkinan *hash value* dari suatu

fungsi *hash* dapat dilihat dari bit yang disediakan sebagai *output* masing-masing fungsi, yang adalah sebagai berikut :

- MD5 : 128 *bit* (2^{128} = 3,4028 × 10^{38} kemungkinan)
- SHA-1 : 160 *bit* (2^{160} = 1,4615 × 10^{48} kemungkinan)
- SHA-256 : 256 *bit* (2^{256} = 1,1579 × 10^{77} kemungkinan)
- CRC32 : 32 *bit* (2^{32} = 4,2949 × 10^{9} kemungkinan)

Jumlah yang cukup meyakinkan untuk mengetahui bahwa kemungkinan *collision* yang ada cukup kecil, bahkan CRC32 yang memiliki "batas" untuk *birthday paradox* terkecil, mampu menampung sekitar 2 milyar (2,1474 × 10^{9}) masukan sebelum akhirnya "terancam" mengalami *collision* sebesar 50%. Sayangnya, jumlah *bit output* dari masing-masing fungsi tersebut berbanding terbalik dengan panjang *short URL* keluarannya, antara lain :

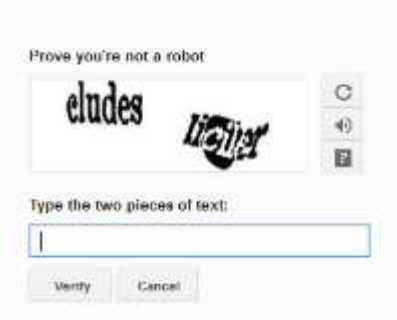
- MD5 (128 *bit*) :
  - 32 karakter basis 16 (*hexadecimal*), atau
  - 26 karakter basis 32 ([a-z]+[0-9], dengan *padding bits*), atau
  - 22 karakter basis 64 ([a-z]+[A-Z]+[0-9]+[+]+[/], dengan *padding bits*).
- SHA-1 (160 *bit*) :
  - 40 karakter basis 16, atau
  - 32 karakter basis 32, atau
  - 27 karakter basis 64 (dengan *padding bits*).
- SHA-256 (256 *bit*) :
  - 64 karakter basis 16, atau
  - 52 karakter basis 32 (dengan *padding bits*), atau
  - 43 karakter basis 64 (dengan *padding bits*).
- CRC32 (32 *bit*) :
  - 8 karakter basis 16, atau
  - 7 karakter basis 32 (dengan *padding bits*), atau
  - 6 karakter basis 64 (dengan *padding bits*).

Bila dibandingkan dengan kemungkinan penggunaan fungsi *hash* seperti yang telah kita katakan diatas, maka :

- Sebagai pembanding alamat *URL* untuk mengidentifikasi duplikasi, tiap *fungsi* dan tiap representasi dapat digunakan dengan baik, namun
- Sebagai *short URL*, hanya CRC32 dengan representasi basis 64 yang mungkin untuk digunakan, bila dibandingkan dengan *shortURL* keluaran goo.gl yang berkisar antara 4-5 karakter, bit.ly yang mengembalikan 6 karakter, dan tinyURL yang mengembalikan 7 karakter.

Apakah fungsi *hash* CRC32 dapat berfungsi dengan baik secara terus-menerus dalam menghadapi perkembangan internet yang cukup pesat? Hasil survey Netcraft mengatakan bahwa jumlah situs yang aktif di internet pada saat ini adalah 644.275.754 buah, dan itu belum termasuk halaman (*webpage*) yang tersedia untuk masing-masing situs tersebut. Bukanlah hal yang mustahil apabila "batas aman" fungsi CRC32 akan terlampaui dalam waktu dekat, atau bahkan sudah terlampaui pada saat ini. Untuk mencegah hal ini terjadi, ada beberapa ide yang cukup mudah untuk diterapkan :

- Penggunan *captcha* untuk mencegah serangan yang dilakukan oleh bot, seperti yang dilakukan oleh goo.gl, atau
- Pemisahan antara *domain* situs dan halamannya (contoh : http://lnk.nu/codinghorror.com/fl2.html), seperti yang diajukan dalam sebuah komentar oleh Greg Hewgill pada [2].



Gambar 3.1*Penggunaan captcha oleh goo.gl untuk mencegah bot spam yang menggunakan layanan URL Shortening Service*

Ide kedua yang diajukan oleh Greg Hewgill terlihat "melanggar" aspek kedua dari *URL Shortening Service*, yakni panjang *short URL* yang cukup pendek, namun dapat mengurangi kemungkinan terjadinya suatu *collision*, dan dapat membantu pengguna untuk mengetahui *domain* website yang hendak mereka tuju (oleh karena format yang diajukan menunjukkan *domain* dari halaman tersebut). Namun, ide "pemisahan *domain* situs dengan halaman" ini dapat diubah sehingga membantu "mencegah" tercapainya jumlah yang melebihi "batas aman" CRC 32, antara lain pembangkitan *short URL* yang terdiri atas konkatenasi (penggabungan dua buah *string*) representasi basis 64 dari:

- *Hash value* dari *domain* situs dan halaman yang dituju, masing-masing diproses dengan fungsi CRC32, dengan panjang *shortURL* sebanyak 12 karakter,
- *Hash value* dari *domain* situs yang diproses dengan fungsi CRC32, dan *hash value* dari halaman yang dituju yang diproses dengan fungsi CRC16, dengan panjang *short URL* sebanyak 9 karakter (*hash value* dari CRC16 dengan representasi basis 64, ditambah dengan *padding bits* mengembalikan 3 karakter)

CRC16 digunakan oleh karena ia memiliki panjang *output* yang cukup kecil (16 *bit*), dan memiliki sebanyak 2^{16} , atau 65.536 kemungkinan, yang merupakan jumlah yang cukup banyak untuk menampung halaman yang

terdapat dalam suatu situs (dan kemungkinan besar dimasukkan pengguna untuk diketahui *short URL* nya). CRC16 tidak digunakan untuk memroses alamat domain *situs*, oleh karena jumlah situs yang ada internet pada saat ini sudah melebihi jumlah kemungkinan *output* dari *hash value* CRC16.

Berikut adalah hasil pengujian *collision* dan kecepatan rata-rata dari kedua metode diatas, dibandingkan dengan metode *hash* lainnya :

Nama Fungsi	Jumlah *collision*	Kecepatan rata-rata
MD5	0	0.0000099829
SHA-1	0	0.0000093321
SHA-256	0	0.0000131795
CRC32	0	0.0000024424
Concat3232	0	0.0004780685
Concat3216	0	0.0003903200

Tabel 3.2 *Hasil pengujian collision dan kecepatan MD5, SHA-1, SHA-256, CRC32, konkatenasi CRC32-CRC32 (Concat3232), serta konkatenasi CRC32-CRC16 (Concat3216) terhadap 4000 buah URL, format keluaran basis 16 Proses konversi hasil hash ke basis 16 juga termasuk dalam perhitungan ini.*

. Dari tabel 3.2, dapat dilihat bahwa kecepatan dari konkatenasi *hash value* dari *domain* dan halaman, masing-masing dengan CRC32, dan konkatenasi *hash value* dari *domain* dengan CRC32 dan halaman dengan CRC16 memiliki perbedaan yang cukup signifikan, yakni sekitar 2×10^2 dari CRC32 saja. Kecepatan tersebut masih termasuk cukup cepat (kurang dari 1 milidetik), dan memiliki hasil pengujian *collision* yang sangat baik, yakni tanpa *collision* sama sekali. Metode ini tidak memberitahu pengguna tentang *domain* situs yang hendak ia tuju seperti komentar yang diajukan oleh Greg Hegwill, dan menghasilkan *short URL* yang lebih panjang daripada *hash value* CRC32 biasa, namun menawarkan ketahanan terhadap *collision* yang lebih baik, dalam perbedaan panjang *y*ang tidak terlalu berlebihan (12 karakter basis 64 untuk konkatenasi CRC32-CRC32, dan 9 karakter basis 64 CRC32-CRC16).

IV. KESIMPULAN

Fungsi *hash* " tradisional" yang selama ini digunakan secara umum, yakni MD5, SHA-1, SHA-256, dan CRC32 dapat diaplikasikan pada sistem pencegahan duplikasi data pada sistem pembangkitan *short URL* untuk *URL Shortening Service* , namun hanya CRC32 saja yang "nyaman" untuk digunakan oleh pengguna untuk pembangkitan *short URL*, karena mengembalikan *hash value* yang cukup pendek, yakni 6 karakter dalam *encoding* basis 64. CRC32 sendiri menawarkan ketahanan terhadap *collision* yang cukup baik, yakni dengan "batas aman" sekitar 2 milyar ($2{,}1474 \times 10^9$), dari total 4 milyar ($4{,}2949 \times 10^9$) kemungkinan. Untuk menambah ketahanan dari fungsi CRC32 ini, sehingga dapat mengikuti perkembangan pertumbuhan situs serta halaman yang terdapat pada internet, terdapat dua metode yang berdasarkan kepada pemisahan alamat *domain* situs dan halaman yang hendak dituju, antara lain pembangkitan *short URL* yang berasal dari konkatenasi dari *hash value* alamat *domain* situs dan *hash value* dari alamat halaman yang dituju, masing-masing dengan fungsi *hash* CRC32, dengan panjang *short URL* sebanyak 12 karakter, dan dari konkatenasi dari *hash value* alamat *domain* situs dengan fungsi CRC32 dengan *hash value* alamat halaman yang dituju dengan fungsi CRC16.

Oleh karena keterbatasan penulis, pengujian yang dilakukan di makalah ini hanya dilakukan dengan 4000 URL yang tersedia, dan tidak meneliti apakah URL yang terdapat di internet dapat menjadi sebuah *string* yang menghasilkan *collision*, seperti serangan *collision* yang dilakukan terhadap fungsi *hash* MD5 pada [6]. Penulis berharap, untuk perkembangan kedepannya, dapat dilakukan pengujian dengan jumlah data uji yang lebih banyak, pengujian serangan *collision*, *working implementation* yang dapat digunakan oleh pengguna internet, dan mungkin fungsi *hash* yang dapat memenuhi aspek-aspek yang diperlukan dari *URL Shortening Service*, yang lebih efektif daripada fungsi *hash* tradisional, yang metodenya dibagikan ke khayalak ramai (*open-source*).

V. UCAPAN TERIMA KASIH

Penulis mengucapkan terima kasih kepada Dr. Rinaldi Munir selaku dosen mata kuliah IF3058 Kriptografi, yang telah mengajarkan dan memberikan bantuan sehingga makalah ini dapat selesai, dan kepada orang-orang yang telah membantu penyelesaian makalah ini, yang tidak dapat penulis sebutkan satu persatu.

REFERENSI

[1] Munir, Rinaldi, *Diktat Kuliah IF5054 Kriptografi*, Penerbit Informatika : Bandung, 2005.

[2] Jeff Atwood, "URL Shortening : Hashes In Practice." Internet : http://www.codinghorror.com/blog/2007/08/url-shortening-hashes-in-practice.html (http://goo.gl/w49V), diakses pada 10 Mei 2012.

[3] Megiddo Nimrod, McCurley Kevin S. , "Efficient retrieval of uniform resource locators", US6957224 , 18 Oktober 2005.

[4] "How URL Shortening Scripts Work." Internet : http://corpocrat.com/2009/09/29/how-url-shortening-scripts-work/ (http://goo.gl/jnFCu), diakses pada 12 Mei 2012.

[5] Jered Floyd, "What do Hash Collisions Really Mean?" Internet : http://permabit.wordpress.com/2008/07/18/what-do-hash-collisions-really-mean/ (http://goo.gl/E32ak), diakses pada 14 Mei 2012.

[6] J. Black, M. Cochran, T. Highland, "A Study of the MD5 Attacks : Insights and Improvements", University of Colorado at Boulder, USA, diterbitkan pada 3 Maret 2006.

[7] Julie Bort, "How Many Websites Are There?", Internet : http://articles.businessinsider.com/2012-03-08/tech/31135231_1_websites-domain-internet (http://goo.gl/f9Mlp), diakses pada 14 Mei 2012.

DAFTAR GAMBAR

Seluruh gambar yang digunakan dalam makalah ini berasal dari penulis, yang berasal dari *screenshot* dari tiap situs yang sudah dituliskan pada tiap keterangan dari masing – masing gambar.

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, 14 Mei 2012

Ricardo Pramana Suranta (13509014)

LAMPIRAN

Source-code yang digunakan dalam makalah ini dapat diunduh pada http://dl.dropbox.com/u/38341960/ArkHash/ArkHash.java, dan data uji yang digunakan dapat diunduh pada http://dl.dropbox.com/u/38341960/ArkHash/testURL.txt .

Berikut adalah adalah *pseudo-code* dari masing –masing fungsi, diambil dari en.wikipedia.org :

1. MD5

```
    //Note: All variables are
    unsigned 32 bits and wrap
    modulo 2^32 when calculating

    var int[64] r, k

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

//Use binary integer part of the
sines of integers (Radians) as
constants:
for i from 0 to 63
    k[i] := floor(abs(sin(i +
1)) × (2 pow 32))
end for
//(Or just use the following
table):
k[ 0.. 3] := { 0xd76aa478,
0xe8c7b756, 0x242070db,
0xc1bdceee }
k[ 4.. 7] := { 0xf57c0faf,
0x4787c62a, 0xa8304613,
0xfd469501 }
k[ 8..11] := { 0x698098d8,
0x8b44f7af, 0xffff5bb1,
0x895cd7be }
k[12..15] := { 0x6b901122,
0xfd987193, 0xa679438e,
0x49b40821 }
k[16..19] := { 0xf61e2562,
0xc040b340, 0x265e5a51,
0xe9b6c7aa }
k[20..23] := { 0xd62f105d,
0x02441453, 0xd8a1e681,
0xe7d3fbc8 }
k[24..27] := { 0x21e1cde6,
0xc33707d6, 0xf4d50d87,
0x455a14ed }
k[28..31] := { 0xa9e3e905,
0xfcefa3f8, 0x676f02d9,
0x8d2a4c8a }
k[32..35] := { 0xfffa3942,
0x8771f681, 0x6d9d6122,
0xfde5380c }
k[36..39] := { 0xa4beea44,
0x4bdecfa9, 0xf6bb4b60,
0xbebfbc70 }
k[40..43] := { 0x289b7ec6,
0xeaa127fa, 0xd4ef3085,
0x04881d05 }
k[44..47] := { 0xd9d4d039,
0xe6db99e5, 0x1fa27cf8,
0xc4ac5665 }
k[48..51] := { 0xf4292244,
0x432aff97, 0xab9423a7,
0xfc93a039 }
k[52..55] := { 0x655b59c3,
0x8f0ccc92, 0xffeff47d,
0x85845dd1 }
k[56..59] := { 0x6fa87e4f,
0xfe2ce6e0, 0xa3014314,
0x4e0811a1 }
k[60..63] := { 0xf7537e82,
0xbd3af235, 0x2ad7d2bb,
0xeb86d391 }

//Initialize variables:
var int h0 := 0x67452301
var int h1 := 0xefcdab89
var int h2 := 0x98badcfe
var int h3 := 0x10325476

//Pre-processing: adding a
single 1 bit
append "1" bit to message
/* Notice: the input bytes are
considered as bit strings,
  where the first bit is the
most significant bit of the
byte.
  In other words: the order of
bits in a byte is BIG ENDIAN,
  but the order of bytes in a
word is LITTLE ENDIAN */
//Pre-processing: padding with
zeroes
append "0" bits until message
length in bits ≡ 448 (mod 512)
append length mod (2 pow 64) to
message
    /* bit (not byte) length of
unpadded message as 64-bit
little-endian integer */

//Process the message in
successive 512-bit chunks:
for each 512-bit chunk of
message
    break chunk into sixteen 32-
bit little-endian words w[j], 0
≤ j ≤ 15
//Initialize hash value for this
chunk:
```

```
    var int a := h0
    var int b := h1
    var int c := h2
    var int d := h3
//Main loop:
    for i from 0 to 63
        if 0 ≤ i ≤ 15 then
            f := (b and c) or
((not b) and d)
            g := i
        else if 16 ≤ i ≤ 31
            f := (d and b) or
((not d) and c)
            g := (5×i + 1) mod
16
        else if 32 ≤ i ≤ 47
            f := b xor c xor d
            g := (3×i + 5) mod
16
        else if 48 ≤ i ≤ 63
            f := c xor (b or
(not d))
            g := (7×i) mod 16
        temp := d
        d := c
        c := b
        b := b + leftrotate((a +
f + k[i] + w[g]) , r[i])
        a := temp
    end for
//Add this chunk's hash to
result so far:
    h0 := h0 + a
    h1 := h1 + b
    h2 := h2 + c
    h3 := h3 + d
end for

var char digest[16] := h0 append
h1 append h2 append h3
//(expressed as little-endian)

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

2. SHA-1

```
Note 1: All variables are
unsigned 32 bits and wrap modulo
2^32 when calculating
Note 2: All constants in this
pseudo code are in big endian.
        Within each word, the
most significant byte is stored
in the leftmost byte position

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 (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:[31]
    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
```

3. SHA-256

```
Note 1: All variables are unsigned 32 bits and wrap modulo 2^32 when calculating
Note 2: All constants in this pseudo code are in big endian
Initialize variables
(first 32 bits of the fractional parts of the square roots of the first 8 primes 2..19):
h[0..7] :=
0x6a09e667, 0xbb67ae85,
0x3c6ef372, 0xa54ff53a,
0x510e527f, 0x9b05688c,
0x1f83d9ab, 0x5be0cd19
Initialize table of round constants
(first 32 bits of the fractional parts of the cube roots of the first 64 primes 2..311):
k[0..63] :=
0x428a2f98, 0x71374491,
0xb5c0fbcf, 0xe9b5dba5,
0x3956c25b, 0x59f111f1,
0x923f82a4, 0xab1c5ed5,
0xd807aa98, 0x12835b01,
0x243185be, 0x550c7dc3,
0x72be5d74, 0x80deb1fe,
0x9bdc06a7, 0xc19bf174,
0xe49b69c1, 0xefbe4786,
0x0fc19dc6, 0x240ca1cc,
0x2de92c6f, 0x4a7484aa,
0x5cb0a9dc, 0x76f988da,
0x983e5152, 0xa831c66d,
0xb00327c8, 0xbf597fc7,
0xc6e00bf3, 0xd5a79147,
0x06ca6351, 0x14292967,
0x27b70a85, 0x2e1b2138,
0x4d2c6dfc, 0x53380d13,
0x650a7354, 0x766a0abb,
0x81c2c92e, 0x92722c85,
0xa2bfe8a1, 0xa81a664b,
0xc24b8b70, 0xc76c51a3,
0xd192e819, 0xd6990624,
0xf40e3585, 0x106aa070,
0x19a4c116, 0x1e376c08,
0x2748774c, 0x34b0bcb5,
0x391c0cb3, 0x4ed8aa4a,
0x5b9cca4f, 0x682e6ff3,
0x748f82ee, 0x78a5636f,
0x84c87814, 0x8cc70208,
0x90befffa, 0xa4506ceb,
0xbef9a3f7, 0xc67178f2
Pre-processing:
append the bit '1' to the message
append k bits '0', where k is the minimum number >= 0 such that the resulting message length (in bits) is modulo 512, minus 64 bits for the length.
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[0..15]
Extend the sixteen 32-bit words into sixty-four 32-bit words:
for i from 16 to 63
s0 := (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor (w[i-15] rightshift 3)
s1 := (w[i-2] rightrotate 17) xor (w[i-2] rightrotate 19) xor (w[i-2] rightshift 10)
w[i] := w[i-16] + s0 + w[i-7] + s1
Initialize hash value for this chunk:
a := h0
b := h1
c := h2
d := h3
e := h4
f := h5
g := h6
h := h7
Main loop:
for i from 0 to 63
s0 := (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
maj := (a and b) xor (a and c) xor (b and c)
t2 := s0 + maj
s1 := (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
ch := (e and f) xor ((not e) and g)
t1 := h + s1 + ch + k[i] + w[i]

h := g
g := f
f := e
e := d + t1
d := c
c := b
b := a
a := t1 + t2
Add this chunk's hash to result so far:
h0 := h0 + a
h1 := h1 + b
```

```
h2 := h2 + c
h3 := h3 + d
h4 := h4 + e
h5 := h5 + f
h6 := h6 + g
h7 := h7 + h
Produce the final hash value
(big-endian):
digest = hash = h0 append h1
append h2 append h3 append h4
append h5 append h6 append h7
```

4. CRC32

```
   5.   Start message with
      11010011101100
      This is first padded with
      zeroes corresponding to
      the bit length n of the
      CRC. Here is the first
      calculation for computing
      a 3-bit CRC :

11010011101100 000 <--- input
left padded by 3 bits
1011               <--- divisor
(4 bits) = x³+x+1
------------------
01100011101100 000 <--- result

If the input bit above the
leftmost divisor bit is 0, do
nothing. If the input bit above
the leftmost divisor bit is 1,
the divisor is XORed into the
input (in other words, the input
bit above each 1-bit in the
divisor is toggled). The divisor
is then shifted one bit to the
right, and the process is
repeated until the divisor
reaches the right-hand end of
the input row. Here is the
entire calculation:

11010011101100 000 <--- input
left padded by 3 bits
1011               <--- divisor
01100011101100 000 <--- result
 1011              <--- divisor
...
00111011101100 000
  1011
00010111101100 000
   1011
00000001101100 000
       1011
00000000110100 000
        1011
00000000011000 000
         1011
00000000001110 000
          1011
00000000000101 000
           101 1
-----------------
00000000000000 100 <---remainder
(3 bits)
```

Since the leftmost divisor bit zeroed every input bit it touched, when this process ends the only bits in the input row that can be nonzero are the n bits at the right-hand end of the row. These *n* bits are the remainder of the division step, and will also be the value of the CRC function (unless the chosen CRC specification calls for some postprocessing).

The validity of a received message can easily be verified by performing the above calculation again, this time with the check value added instead of zeroes. The remainder should equal zero if there are no detectable errors.

```
11010011101100 100 <--- input
with check value
1011               <--- divisor
01100011101100 100 <--- result
 1011              <--- divisor
...
00111011101100 100

......

00000000001110 100
          1011
00000000000101 100
           101 1
-----------------
                 0 <---
remainder
```