Penggunaan CRC32 dalam Integritas Data

Indra Sakti Wijayanto
13504029

Program Studi Teknik Informatika
Institut Teknologi Bandung
Jalan Ganesha 10 Bandung 40132

E-mail:
If14029@students.if.itb.ac.id

Abstraksi

Cyclic Redundancy Check (CRC) adalah salah satu fungsi hash yang dikembangkan untuk mendeteksi kerusakan data dalam proses transmisi ataupun penyimpanan. CRC menghasilkan suatu *checksum* yaitu suatu nilai dihasilkan dari fungsi hash-nya, dimana nilai inilah yang nantinya digunakan untuk mendeteksi error pada transmisi ataupun penyimpanan data. Nilai CRC dihitung dan digabungkan sebelum dilakukan transmisi data atau penyimpanan, dan kemudian penerima akan melakukan verifikasi apakah data yang diterima tidak mengalami perubahan ataupun kerusakan. CRC cukup terkenal karena mudah diterapkan dalam hardware, dan mudah dilakukan analisis secara matematika. Prinsip utama yang digunakan adalah dengan melakukan pembagian polinomial dengan mengabaikan bit-bit carry. Cara yang biasa digunakan adalah dengan menggunakan tabel CRC yang nilainya telah dihitung sebelumnya, sehingga dapat menghemat waktu dan meminimalisir kesalahan di tengah perhitungan.

Meskipun CRC termasuk fungsi hash, tetapi CRC tidak cukup aman karena telah ditemukan cara untuk melakukan reversing terhadap hasil CRC. Kemampuan untuk melakukan reverse terhadap nilai CRC ini dapat kita manfaatkan ketika kita ingin melakukan manipulasi terhadap data yang kita ketahui nilai CRCnya, misalnya dalam proses kompresi data, dimana kita tidak perlu mengubah nilai CRC (dengan menghitung ulang keseluruhan data). Karena cukup flexible, algoritma CRC banyak diterapkan dalam beberapa bahasa pemorograman misalnya PHP, Java, C, Perl, dll. Varian CRC yang saat ini banyak digunakan adalah CRC32.

Kata kunci : *Cyclic Redundancy Check, checksum.*

1. Pendahuluan

Proses pengiriman ataupun penyimpanan data seringkali mempunyai resiko terjadi perubahan yang tidak diinginkan terhadap data. Hal ini seringkali terjadi pada level fisik (media atau saluran yang digunakan), yang disebabkan karena gangguan (*noisy*) pada proses penyimpanan ataupun pengiriman data itu sendiri. Untuk mendeteksi kerusakan data ini, digunakan suatu cara untuk menghitung suatu nilai terhadap data yang diberikan, dan nilai tersebut dikirim bersama-sama data untuk dicek oleh penerima apakah data yang diterima sama dengan aslinya (tidak mengalami kerusakan selama perjalanan atau penyimpanan). Kerusakan data ini biasanya hanya terjadi pada satu atau dua bit saja, maka untuk menghitung nilai data tersebut tidak perlu digunakan suatu fungsi hash yang benar-benar aman (rumit). Hal ini karena tujuan kita bukanlah melindungi data dari *cracker*, melainkan untuk menjaga integritas data saat proses *transmition* atau *storage*.

Salah satu fungsi hash yang biasa digunakan adalah CRC (*Cyclic Redundancy Check*) yang mempunyai beberapa varian bergantung pada bilangan *polynomial* yang digunakan dalam proses komputasinya. Meskipun CRC tidak cukup aman sebagai algoritma kriptografi, tetapi CRC cukup efektif

digunakan karena mudah diimplementasikan dan cukup cepat dalam melakukan komputasinya.

Dalam makalah ini, penulis akan mencoba untuk menguraikan beberapa hal penting yang digunakan dalam melakukan penghitungan nilai CRC dan melakukan contoh penghitungan secara sederhana, baik menggunakan pendekatan aljabar maupun menggunakan tabel CRC yang telah terdefinisi. Makalah ini juga akan menguraikan prinsip yang digunakan dalam melakukan reversing terhadap suatu nilai CRC yang telah diketahui sehingga dapat bermanfaat pada saat modifikasi data.

2. Cara kerja (prinsip) CRC

Pada intinya dalam proses penghitungan CRC, kita menganggap suatu file yang kita proses sebagai suatu string yang besar, yang terdiri dari bit-bit, dan kita operasikan sebagai suatu bilangan *polynomial* yang sangat besar. Untuk menghitung nilai CRC, kita **membagi** bilangan polynomial, sebagai representasi dari file, dengan suatu bilangan polynomial kecil yang sudah terdefinisi untuk jenis varian CRC tertentu. Nilai CRC adalah **sisa** hasil bagi tersebut, yang biasa disebut dengan **checksum**.

Setiap operasi pembagian pasti menghasilkan suatu sisa hasil bagi (meskipun bernilai 0), tetapi ada perbedaan dalam melakukan pembagian pada penghitungan CRC ini. Secara umum (prinsip aljabar biasa), pembagian dapat kita lakukan dengan mengurangi suatu bilangan dengan pembaginya secara terus-menerus sampai menghasilkan suatu sisa hasil bagi (yang lebih kecil dari bilangan pembagi). Dari nilai hasil bagi, sisa hasil bagi, dan bilangan pembagi kita bisa mendapat bilangan yang dibagi dengan mengalikan bilangan pembagi dengan hasil bagi dan menambah dengan sisa hasil bagi.

Dalam penghitungan CRC, operasi pengurangan dan penjumlahan dilakukan dengan mengabaikan setiap nilai *carry* yang didapat. Tentu saja hal ini juga akan berpengaruh pada proses pembagian yang menjadi dasar utama dalam melakukan penghitungan nilai CRC. Operasi dalam CRC juga hanya melibatkan nilai 0 dan 1, karena secara umum kita beropersi dalam level bit. Contoh penghitungan dalam CRC adalah sebagai berikut:

```
(1) 1101  (2) 1010  1010
    1010-     1111+ 1111-
    ----      ----  ----
    0011      0101  0101
```

Pada contoh tersebut, operasi pertama (1) adalah operasi yang umum digunakan dalam operasi aljabar, yaitu dengan menghitung nilai *carry* yang dihasilkan, sedangkan operasi kedua (2) adalah operasi dasar yang akan kita gunakan dalam proses penghitungan nilai CRC. Nilai carry diabaikan, sehingga operasi pengurangan dan penambahan akan menghasilkan suatu nilai yang sama. Kedua operasi ini bisa dilakukan dengan melakukan penjumlahan dan dimodulo 2, atau dalam dunia pemrograman lebih dikenal dengan operasi XOR (istilah ini yang akan lebih sering kita gunakan untuk menyebut penjumlahan pada operasi penghitungan CRC).

Pada proses pembagian yang dilakukan, akan tampak sekali bedanya karena pengurangan yang dilakukan dilakukan seperti melakukan penambahan.

Nilai hasil bagi diabaikan, karena kita tidak menggunakannya, jadi hanya sisa hasil bagi (*remainder*) yang kita perhatikan. Dan *remainder* inilah yang akan menjadi dasar bagi nilai CRC yang dihasilkan.

3. Penghitungan CRC Secara Aljabar

Untuk melakukan penghitungan CRC terhadap suatu data, maka yang pertama kita perlukan adalah suatu bilangan polinom yang akan menjadi pembagi dari data yang akan kita olah (kita sebut sebagai ‘poly’). Kita juga menghitung lebar suatu poly, yang merupakan posisi dari bit tertinggi. Misalkan kita sebut lebar dari poly ini adalah W, maka jika kita mempunyai poly 1001, maka W poly tersebut adalah 3, bukan 4. Bit tertinggi ini harus kita pastikan bernilai 1. mengetahui nilai W secara tepat sangat penting karena akan berpengaruh pada jenis CRC yang kita gunakan (CRC16,CRC32,dll).

Data yang kita olah mungkin saja hanya beberapa bit saja, lebih kecil dari nilai poly yang kita gunakan. Hal ini akan menyebabkan kita tidak mengolah semua nilai poly yang telah ditentukan. Untuk mengatasi hal tersebut, dalam penghitungan dasar secara aljabar, kita **menambah** suatu string bit sepanjang W pada

data yang akan kita olah, untuk menjamin keseluruhan data kita proses dengan benar.
Contoh penghitungan kita menjadi sebagai berikut:

```
Poly                  = 10011
(width W=4)
Bitstring + W zeros = 110101101
+ 0000
Contoh pembagian yang dilakukan:

10011/1101011010000\110000101
      10011|||||||| -
      -----||||||||
       10011|||||||
       10011|||||||  -
       -----|||||||
        00001||||||
        00000||||||   -
        -----||||||
         00010|||||
         00000|||||    -
         -----|||||
          00101||||
          00000||||     -
          -----||||
           01010|||
           00000|||      -
           -----|||
            10100||
            10011||       -
            -----||
             01110|
             00000|        -
             -----|
              11100
              10011         -
              -----
               1111 -> sisa hasil bagi
```

Nilai remainder inilah yang menjadi nilai CRC. Pada proses pembagian tersebut, kita mendapat hal penting yang perlu kita perhatikan dalam penghitungan secara aljabar ini adalah kita tidak perlu melakukan operasi XOR ketika bit tertinggi bernilai 0, tapi kita hanya melakukan penggeseran (*shift*) sampai didapat bit tertinggi yang bernilai 1. Hal ini akan sedikit mempermudah dan mempercepat operasi aljabar kita.
Secara notasi Aljabar bisa kita tuliskan sebagai berikut:

```
a(x).xN = b(x).p(x)+r(x)
```

keterangan :

a(x)	:	Bilangan polynomial yang merepresentasikan data.
xN	:	Nilai 0 sebanyak W
b(x)	:	hasil bagi yang didapat
p(x)	:	poly
r(x)	:	sisa hasil bagi, nilai CRC

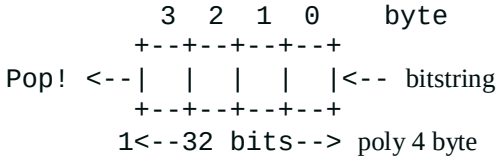
karena nilai CRC adalah sisa hasil bagi, maka untuk mengecek integritas data dapat dilakukan dengan beberapa cara, diantaranya:

a. Kita hitung nilai CRC dari data yang asli, lalu kita cocokkan dengan nilai CRC yang disimpan (di *append* dengan data). Data yang asli mudah kita dapatkan karena nilai CRC sepanjang N-1.
b. Data yang asli kita tambah dengan nilai CRC, lalu kita bagi dengan nilai poly, maka sisa hasil bagi adalah 0 jika data benar.

4. Pendekatan Tabel CRC

Penghitungan nilai CRC yang berbasis bit seperti pada pendekatan aljabar diatas akan sangat lama dan tidak *efficient*. Kita bisa memperbaiki cara yang kita gunakan jika kita dapat melakukan operasi dengan basis **byte**, bukannya **bit**. Poly yang kita gunakan pun akan kita operasikan dalam bentuk byte, sehingga harus mempunyai panjang kelipatan 8 bit (byte).

Dalam CRC32, berarti kita gunakan poly 32 bit (4 byte), akan tampak sebagai berikut:



4 ruang kosong pada ilustrasi diatas menggambarkan register yang akan kita gunakan untuk menampung hasil CRC sementara (pada proses pembagian yang melibatkan operasi XOR). Proses yang dilakukan pada register ini adalah :

a. Kita masukkan bitstring (data) ke dalam register dari arah ke kanan (*shift* per byte) setiap saat register tidak terisi penuh.
b. Jika register sudah penuh (berisi 4 byte), maka kita geser satu byte ke arah kiri (keluar register), dan kita isi register dari arah kanan dengan 1 byte dari bitstring
c. Jika register yang digeser keluar punya nilai 1, maka kita melakukan operasi XOR

terhadap keseluruhan isi register (termasuk yang telah digeser keluar, dimulai dari bit tertinggi yang bernilai 1) dan nilai dari poly. Kita ulang langkah ini sampai semua bit dari byte yang tergeser keluar bernilai 0.

d. Kita ulang langkah b dan d sampai semua bitstring (data input) selesai kita proses.

Operasi yang kita lakukan diatas pada intinya sama dengan operasi aljabar yang kita lakukan, hanya kita melakukan dengan pendekatan byte.

Sebagai contoh, kita akan memproses CRC 8 bit (poly hanya 1 byte) agar lebih sederhana. Register akan berisi 8 bit, dan pada sekali *shift* kita akan menggeser sebanyak 4 bit.

Isi awal Register : 1011**0100**

Kemudian kita akan menggeser keluar 4 bit pertama (1011), dan memasukkan 4 bit baru misalkan 1101.

Maka, 8 bit dalam register adalah : **0100**1101
4 bit yang digeser (top bits) : 1011
Poly yang kita gunakan (W=8) : 101011100

Langkah selanjutnya yang kita lakukan adalah melakukan XOR terhadap isi register dengan poly.

```
Top  Register
---- --------
1011 01001101
1010 11100   +(*1)Operasi Xor dimulai
-------------  pada bit tertinggi bernilai 1
0001 10101101 hasil XOR
```

Karena Top (isi register yang digeser keluar) masih mengandung nilai 1, maka kita ulangi lagi langkah ini.

```
0001 10101101   hasil XOR sebelumnya
   1 01011100+(*2)Operasi Xor dimulai
-------------  pada bit tertinggi bernilai 1
0000 11110001  hasil akhir XOR
^^^^
Nilai pada register: 11110001
```

Karena semua bit pada Top Register telah bernilai 0, maka kita telah menyelesaikan satu putaran untuk memproses rangkaian bit yang kita geser keluar.

Karena secara aritmatik operasi XOR bersifat komutatif, maka kita akan mendapatkan hasil yang sama pada register jika kita melakukan operasi XOR pada (*1) dan (*2) terlebih dahulu, kemudian hasilnya kita XOR dengan isi register.

```
1010 11100     (*1)
   1 01011100+ (*2)
-------------
1011 10111100  (*3) hasil XOR awal

nilai (*3) kita XOR dengan isi
Register, maka:

1011 10111100 (*3)
1011 01001101+ keseluruhan isi register
-------------
0000 11110001 à isi register
```

Pendekatan kita yang terakhir akan sangat berguna karena untuk nilai poly tertentu, dan top bits (nilai yang digeser keluar) tertentu, maka nilai (*3) pasti dapat kita hitung dulu, yaitu kita XOR sampai semua top bits bernilai 0. hal ini akan sangat berguna bagi kita,karena kombinasi top bits dapat kita perkirakan sebelumnya, maka dengan melakukan komputasi awal terhadap semua kemungkinan top bits, kita dapat semua nilai (*3) yang mungkin.

Pada CRC 32, untuk setiap **byte** yang digeser keluar kita bisa menghitung nilai yang akan kita gunakan dalam operasi XOR dengan isi register. Nilai-nilai ini akan kita simpan, dan kita sebut dengan tabel CRC. Tabel ini mempunyai index yang merupakan nilai top bits, yang nilai isi dari tabel mengacu pada nilai (*3).

Contoh kode (dalam C) untuk membentuk tabel CRC sebagai berikut:

```
/**
* menghasilkan CRC table dengan 256
32- bit entries .
Prekondisi table telah dialokasi
dengan jumlah yang tepat (256 entri)
*/

void make_table ( uint32 * table ) {
   uint32 c;
   int n, k;
   for (n = 0; n < 256; n ++) {
      c = n;
      for(k = 0; k < 8; k ++) {
            if ((c & 1) != 0) {
            c = poly ^ (c >> 1);
            //poly telah diketahui
            }
            else {
                  c = c >> 1;
            }
      }
      table [n] = c;
}
}
```

Dengan pendekatan tabel CRC ini, algoritma register CRC menjadi:

```
while (masih_ada_bitstring)
    begin
      Top=Top_byte_of_Reg
      Reg=Reg<<8 {geser 1 byte}
      Reg=Reg or new_bytestring
            {tambah byte baru}
      Reg=Reg xor Table[Top]
      {xor dengan nilai dari
tabel dengan index Top}
    End
```

5. Algoritma Direct Table

5.1 Konsep Direct Table

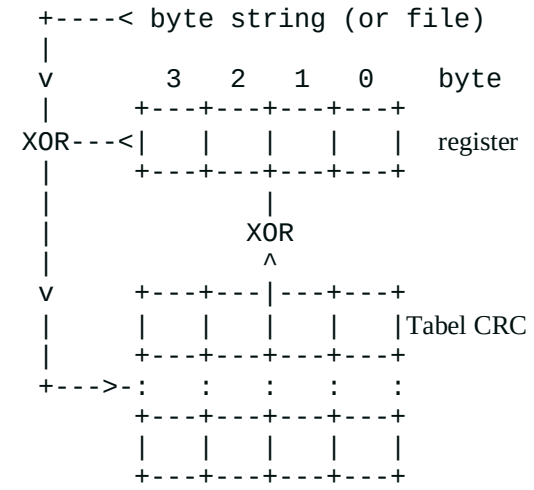
Algoritma yang kita gunakan pada pendekatan diatas, dapat kita optimasi lagi jika kita bisa mencegah satu byte dari bytestring melewati setiap bagian dari register (konsekuensi dari penggeseran 1 byte) sebelumnya akhirnya diproses untuk mengacu nilai pada tabel CRC.

Pada algoritma direct table, kita melakukan operasi XOR secara langsung sebuah byte dari bytestring dengan byte yang *dishift out* dari register (register tidak diisi dengan byte dari bytestring). Hasil XOR ini akan menjadi index pada tabel, yang nilainya akan dilakukan XOR dengan isi dari register.

5.2 Algoritma Direct Table

Untuk melakukan algoritma ini, kita menginisiasi register dengan suatu nilai, kita sebut INITXOR (biasanya berupa bilangan dalam hexa FFFFFFFF). Kita juga tidak melakukan penggeseran terhadap nilai bytestring ke dalam register. Nilai register hanya didapat pada setiap operasi Xor yang dilakukan terhadap isi awal register dengan isi tabel CRC. Dengan adanya INITXOR yang memberikan nilai awal dari register, kita tidak membutuhkan bilangan 0 sejumlah W yang harus kita *append* pada bytestring, dan sebagai konsekuensinya pada akhir perhitungan nilai CRC (*checksum*) yang kita dapatkan juga akan kita XOR dengan suatu bilangan hexa, yang kita kita sebut FINALXOR dan biasanya berupa bilangan FFFFFFFF.

Gambaran Algoritma:



keterangan:

- Ambil satu byte dari register, misal b1
- XOR b1 dengan satu byte dari data, misalkan hasilnya b2.
- Gunakan b2 sebagai index untuk mendapatkan nilai pada tabel CRC, misalkan nilai yang didapat b3.
- Lakukan XOR b3 dengan isi register, dan hasilnya dimasukkan dalam register menggantikan isi register awal.

6. Reflected Direct Table

6.1 Konsep Reflected Direct Table

Algoritma direct table yang dikembangkan lebih jauh lagi dan lebih rumit menggunakan prinsip *reflection.* Sebagai contoh *reflection*, 0111011001 adalah hasil *reflection* dari 1001101110 (yang kita lakukan hanyalah menggunakan prinsip pencerminan dengan pusat bit di tengah).

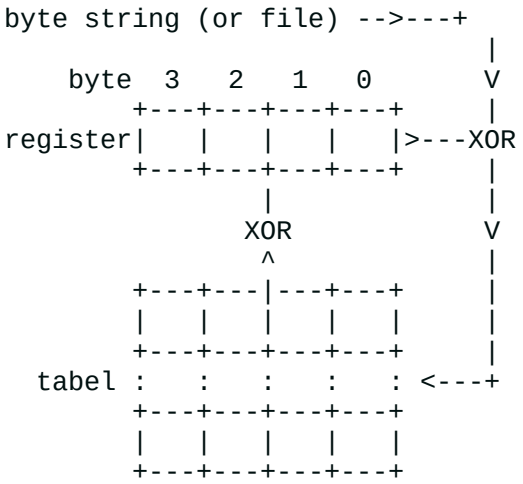
Kita gunakan prinsip *reflection* untuk mengembangkan algoritma tabel CRC kita karena chip yang digunakan untuk operasi IO (UART) mengirimkan setiap byte dengan LSB (least significant bit, bit ke0) dikirim terlebih dahulu, dan MSB (Most Significant Bit) dikirim terakhir. Hal ini merupakan kebalikan dari kondisi yang secara umum berlaku.

Untuk mengatasi hal ini, kita tidak perlu melakukan *reflecting* terhadap bytes data yang akan kita proses. Yang perlu kita lakukan hanyalah melakukan pergeseran ke

kanan (pada algoritma biasa kita melakukan pergeseran kea rah kiri) pada register CRC dan membaca tabel dengan cara yang sama.

6.2 Algoritma

Secara umum, cara yang digunakan sama dengan algoritma biasa, kecuali pergeseran pada register yang dibalik.
Ilustrasi proses ini:



Keterangan:

Karena byte yang datang dari LSB (bit ke-0 lebih dulu) maka pemrosesan kita lakukan dari arah kanan (dibalik).

Dengan cara ini, Algoritma reflected direct table menjadi sebagai berikut:

a. Geser ke kanan isi dari register sebanyak 1 byte. Misalkan kita dapat byte b1.
b. Lakukan operasi XOR terhadap b1 dengan satu byte baru dari data yang akan kita proses. Misalkan hasil operasi XOR ini byte b2.
c. Byte b2 kita gunakan sebagai index untuk mencari nilai tabel yang akan kita gunakan untuk operasi XOR dengan isi keseluruhan register. Kita ambil isi dari table[b2] dan kita XOR-kan dengan isi register, kemudian hasil operasi XOR ini akan menggantikan isi register semula.
d. Ulangi langkah a sampai c sampai semua byte data telah diproses. Sisa hasil bagi (remainder) adalah isi dari register yang terakhir.

6.3 Impelementasi

Untuk menerapkan konsep dan algoritma diatas yang menggunakan tabel CRC, kita memerlukan strandar yang digunakan dalam penghitungan CRC, sebagai berikut:

Standar CRC32

Name	: "CRC-32"
Width	: 32
Poly	: 04C11DB7 **à** Hexa Atau $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ **à** Polinom
Initial (INITXOR)	: FFFFFFFF
Reflected	: True
FINALXOR	: FFFFFFFF

Standar CRC16

Name	: "CRC-16"
Width	: 16
Poly	: 8005 **à** Hexa Atau $x^{16} + x^{15} + x^2 + 1$ **à** Bentuk Polinom
Initial (INITXOR)	: 0000
Reflected	: True
FINALXOR	: 0000

INITXOR adalah nilai awal yang diisikan pada register, pada CRC32 berisi 4 byte, dan 2 byte pada CRC16. jumlah ini sesuai dengan jumlah angka 0 yang harus ditambahkan jika kita menggunakan operasi aljabar biasa. FINALXOR adalah suatu nilai yang ditambahkan (dilakukan operasi XOR) pada *remainder* / sisa hasil bagi pada proses penghitungan nilai CRC. Hasil penambahan inilah yang menjadi nilai CRC sesungguhnya (yaitu nilai yang akan kita proses lebih lanjut).

CRC cukup sederhana dan mudah diterapkan di level *hardware*, hal ini karena operasi utama yang digunakan hanyalah operasi XOR, pergeseran bit, dan perbandingan. Operasi-operasi tersebut cukup efektif diterapkan di terapkan pada *hardware* dan bisa meningkatkan kecepatan komputasi.

Berikut contoh kode dalam Assembly untuk melakukan penghitungan table CRC dan penggunaaanya.

#Membentuk Tabel CRC

```
xor ebx, ebx    ;ebx=0

InitTableLoop:
xor eax, eax    ;eax=0 for new entry
mov     al, bl
;lowest 8 bits of  ebx are copied into lowest 8
;bits of eax

;generate entry
xor     cx, cx

entryLoop:
        test    eax, 1
        jz      no_topbit
        shr     eax, 1
        xor     eax, poly
        jmp     entrygoon
 no_topbit:
        shr     eax, 1
 entrygoon:
        inc     cx
        test    cx, 8
        jz      entryLoop

        mov  dword ptr[ebx*4 +
crctable], eax
        inc     bx
        test    bx, 256
        jz      InitTableLoop
```

#Implementasi table CRC

```
computeLoop:
        xor     ebx, ebx
        xor     al, [si]
        mov     bl, al
        shr     eax, 8
        xor     eax, dword ptr
[4*ebx +crctable]

        inc     si
        loop   computeLoop

        xor     eax, 0FFFFFFFFh
```

Notes: - ds:si points to the buffer where the bytes to process are

- cx contains the number of bytes to process
- eax contains current CRC
- crctable is the table computed with the code above
- the initial value of the CRC is in the case of CRC-32: FFFFFFFF
- after complete calculation the CRC is XORred with: FFFFFFFF

which is the same as NOTting.

(ditulis oleh Anarchriz, dalam **http://www.woodmann.com/fravia/crctut1.htm**)

7. Reversing CRC

7.1 Ide Awal

Dalam banyak keadaan, kita mungkin mendapatkan suatu data yang di dalamnya sudah mempunyai nilai CRC. Kita ingin memodifikasi data tersebut (menambah, mengubah, atau bahkan mengurangi) tetapi kita tidak ingin mengubah nilai CRC yang sudah ada. Hal ini karena menghitung nilai CRC yang baru akan membutuhkan waktu lebih banyak, dan yang lebih penting file tersebut bisa jadi hanya dapat diautentikasi atau bahkan hanya bisa diproses dengan nilai CRC yang sudah diberikan (nilai CRC tertentu).

Dalam hal ini kita bisa melakukan sedikit manipulasi terhadap data kita, sehingga kita tidak perlu mengubah nilai CRC yang sudah dihitung sebelumnya. Jika kita melakukan perubahan data di tengah file, maka yang kita perlukan adalah serangkaian byte (jumlahnya tergantung pada jenis CRC) yang akan mengembalikan isi register CRC pada posisi setelah perubahan data. Hal ini penting agar proses penghitungan nilai CRC pada akhirnya tetap menghasilkan nilai yang sama. Misalkan kita ingin mengubah byte pada posisi 10 sampai dengan 20, maka kita perlu memanipulasi byte 21 sampai dengan 24 sehingga nilai register pada saat penghitungan CRC tetap sama ketika mencapai posisi byte 25 dan seterusnya. Hal ini jika kita menggunakan CRC32, dimana kita membutuhkan 4 byte tertentu untuk mengubah isi suatu register menjadi isi yang berbeda yang telah kita ketahui sebelumnya. Sebagai gambaran, langkah umum yang kita perlukan jika kita ingin mengubah byte pada posisi 10 sampai dengan 20 adalah sebagai berikut:

a. kita hitung nilai register CRC dari byte pertama sampai dengan byte ke 9. simpan hasinya, misalnya kita beri nama R0.

b. Kita lanjutkan menghitung nilai register CRC dari posisi 9 sampai dengan posisi 20, dan dilanjutkan lagi sampai posisi 24. kita dapatkan suatu nilai register CRC. Nilai register inilah yang juga harus kita dapatkan setelah menghitung sampai byte 24 ketika kita sudah mengubah byte 10 sampai 20. Kita simpan nilai register ini, misalkan kita beri nama Original_Register.
c. Kita ubah nilai byte dari posisi 10 sampai byte 20 dengan byte-byte baru yang kita perlukan. Lalu kita hitung nilai Register CRC, dimulai dari R0 (sebagai register awal,menggantikan INITXOR) sampai byte ke 20. kita catat hasilnya misalnya sebagai New_Register.
d. Kita hitung 4 nilai byte baru dari posisi 21 sampai 24, sehingga dari nilai New_Register kita bisa mendapatkan Original_Register dengan menggunakan 4 byte baru ini. Dengan langkah ini, maka nilai akhir CRC akan tetap. Kita bisa menjamin hal ini dengan catatan nilai byte setelah posisi 24, tidak mengalami perubahan (masih sama).

Untuk menghitung 4 nilai byte baru (pada langkah d.) kita gunakan algoritma reversing CRC, sesuai jenis yang digunakan. Penjelasan lebih lengkap dalam sub bab dibawah.

7.2 **Reversing CRC16**

Untuk memulai memlakukan algoritma reversing, prekondisi yang diperlukan adalah nilai register awal dan nilai register akhir yang ingin kita dapat. Dalam hal ini kita ingin mencari byte baru yang diperlukan untuk mengubah nilai register awal menjadi register akhir. Pada sub bab ini, penulis membahas terlebih dahulu algoritma yang berlaku pada CRC16 sehingga jumlah byte yang diperlukan adalah 2 byte (register CRC juga berisi 2 byte)

Langkah yang kita lakukan dengan memanfaatkan induksi matematika adalah sebagai berikut:

a. Kita beri nama 2 byte baru yang ingin kita cari dengan nama X dan Y. Sedangkan register awal kita beri nilai awal byte a1 dan a0.
b. 2 byte string X Y kita baca dari kiri kekanan, sedangkan isi register kita baca dari kanan (a0 dulu baris a1).
c. Kita proses byte pertama (X), maka akan kita dapatkan:
   - a0 + X (byte kita XOR dengan isi register yang di *shift* ke kanan, misalkan hasil XOR adalah i)
   - b1 b0 (kita daptkan suatu *sequence* dari isi table CRC dengan index i)
   - 00 a1 (isi register setelah sekali pergeseran ke kanan, sebelum di XOR dengan isi table CRC)
   - 00+b1 a1+b0 (isi register setelah dilakukan operasi XOR dengan isi tabel[i])

   Isi register terakhir setelah memproses byte pertama adalah (`b1`) dan (`a1+b0`)

d. Kita proses byte kedua (Y) dengan cara yang sama, maka akan kita dapatkan:
   - (`a1+b0`) + Y (byte kita XOR dengan isi register yang di *shift* ke kanan, misalkan hasil XOR adalah j)
   - c1 c0 (kita daptkan suatu *sequence* dari isi table CRC dengan index j)
   - 00 b1 (isi register setelah sekali pergeseran ke kanan, sebelum di XOR dengan isi table CRC)
   - 00+c1 b1+c0 (isi register setelah dilakukan operasi XOR dengan isi tabel[i])

   Isi register terakhir setelah memproses byte pertama adalah (`c1`) dan (`b1+c0`)

   Secara matematik langkah c dan d dapat kita jabarkan sebagai berikut:

`a0 + X=(1)` hasilnya sebagai index untuk menunjuk b1 dan b0

`a1 + b0 + Y =(2)` hasilnya sebagai index untuk menunjuk c1 dan c0

`b1 + c0=d0` bit kanan register baru

`c1=d1` bit kiri register baru

`(1) (2)`

e. Selanjutnya, karena kita mengetahui nilai d1 dan d0 (nilai

Original_Register), maka kita bisa mengetahui nilai dari c1. Dengan memanfaatkan table CRC, maka kita bisa mengetahui nilai c0 dengan cara mencari nilai c1 sebagai byte awal. Dalam tabel CRC yang kita bentuk kita menjamin bahwa semua isi tabel unik dimulai dari byte awal.

f. Setelah kita mengetahui nilai c0, kita bisa mengetahui nilai b1. dengan rumus b1=d0+c0 (operasi + dan – berarti sama, yaitu operasi XOR). Dengan nilai b1 ini, kita bisa mengetahui nilai b0 dengan menggunakan tabel CRC, dengan cara yang sama dengan langkah e.

g. Setelah kita mengetahui nilai b1 b0 dan c1 c0, maka semua variable kita telah lengkap. Kita dapat dengan mudah menghitung nilai byte X dan Y dengan cara sebagai berikut:

`a1+b0+Y=(2),` maka `Y=a1+b0+(2)`
`a0+X=(1)  ,` maka `X=a0+(1)`

Dengan langkah terakhir inilah kita mendapatkan nilai X dan Y yang akan mengubah New_Register kita (a1 a0) menjadi Original_Register(d1 d0).

Contoh Penggunaan reversing CRC-16 untuk mencari 2 byte baru sebagai berikut (permisalan dengan nilai register yang sebenarnya):

- register awal `:(a1=)DE`
  `(a0=)AD`

- register akhir `:(d1=)12`
  `(d0=)34`

- dengan mengikuti langkah (e), maka kita dapatkan nilai c1=12. Dan setelah melihat pada tabel CRC maka nilai yang diawali dengan 12 adalah 12C0, dan hanya ini nilai yang diawali dengan angka 12. Dari sini, kita bisa mendapatkan nilai c0 yaitu C0.
- dengan mengikuti langkah (f), kita dapatkan nilai b1=34 + C0 = F4.(ubah dalam bentuk bit, kemudian lakukan operasi XOR). Dari nilai b1 ini kita bisa mendapatkan nilai b0 dengan mencari nilai pada table CRC yang mempunyai awalan F4, maka kita dapatkan nilai F441, maka nilai b0 adalah 41.
- semua variable telah kita ketahui, maka untuk menghitung nilai X dan Y, yaitu dengan :
  X = AD+4F = **E2**
  Y = DE+41+38 = **A7**

Dari Contoh diatas, dapat kita simpulkan bahwa untuk mengubah nilai register `DEAD` menjadi `1234` kita perlukan 2 byte bernilai `E2 A7` (dengan urutan tidak boleh terbalik).

7.3 **Reversing CRC32**

Secara umum, cara melakukan reverse terhadap CRC 32 sama dengan cara yang diterapkan pada CRC16, hanya disini kita menggunanakan 4 byte baru dimana pada CRC16 hanya digunakan 2 byte baru.

Untuk memulai melakukan reversing, seperti cara yang kita gunakan pada CRC18, kita memerlukan 4 byte-string, misalkan X Y Z W. kemudian kita misalkan register sudah terisi (terinisialisasi) dengan nilai a3 a2 a1 a0.

Langkah yang kita lakukan dengan perhitungan secara manual, sama seperti dengan pengolahan pada CRC16, sebagai berikut:

Proses setiap byte dimulai byte pertama, X, kemudian kita melakukan ***shift*** kekanan 1 byte terhadap isi register, kemudian kita XOR dengan X, hasilnya kita gunakan sebagai index untuk mengacu pada table CRC. Nilai table CRC yang dihasilkan di-XOR dengan isi register, kemudian hasilnya dimasukkan dalam register. Kita ulang langkah tersebut sampai semua byte (4 byte, yaitu X Y Z W) selesai kita proses.

Langkah setiap pengolahan byte menghailkan sebagai berikut:

Proses byte pertama, X:

XOR X dengan top byte pada register, misal hasilnya (1)
à `a0+X`

Nilai yang ditunjuk pada tabelCRC[(1)]
à `b3 b2 b1 b0`

isi register setalah ***shift*** 1 byte
à `00 a3 a2 a1`

XOR isi register dengan nilai table[(1)]
à `00+b3 a3+b2 a2+b1 a1+b0`

Isi Register terakhir, setelah proses byte X adalah:
(b3) (a3+b2) (a2+b1) (a1+b0)

Proses byte kedua, Y:

XOR Y dengan top byte pada register, misal hasilnya (2)
à (a1+b0)+Y

Nilai yang ditunjuk pada tabelCRC[(2)]
àc3 c2 c1 c0

isi register setalah *shift* 1 byte
à00 b3 a3+b2 a2+b1

XOR isi register dengan nilai table[(2)]
à00+c3 b3+c2 a3+b2+c1 a2+b1+c0

Isi Register terakhir, setelah proses byte Y adalah:
(c3)(b3+c2)(a3+b2+c1)(a2+b1+c0)

Proses byte kedua, Z:

XOR Z dengan top byte pada register, misal hasilnya (3)
à (a2+b1+c0)+Z

Nilai yang ditunjuk pada tabelCRC[(3)]
à d3 d2 d1 d0

isi register setalah *shift* 1 byte
à00 c3 b3+c2 a3+b2+c1

XOR isi register dengan nilai table[(2)]
à00+d3 c3+d2 b3+c2+d1 a3+b2+c1+d0

Isi Register terakhir, setelah proses byte Z adalah:
d3) (c3+d2) (b3+c2+d1) (a3+b2+c1+d0)

Proses byte kedua, W:

XOR W dengan top byte pada register, misal hasilnya (4)
à(a3+b2+c1+d0)+W

Nilai yang ditunjuk pada tabelCRC[(4)]
à e3 e2 e1 e0

isi register setalah *shift* 1 byte
à00 d3 c3+d2 b3+c2+d1

XOR isi register dengan nilai table[(2)]
à00+e3 d3+e2 c3+d2+e1 b3+c2+d1+e0

Isi Register terakhir, setelah proses byte W adalah:
(e3) (d3+e2) (c3+d2+e1) b3+c2+d1+e0)

Dalam notasi matematika, secara lebih ringkas bisa kita tuliskan sebagai berikut:

```
a0 + X                    =(1)
a1 + b0 + Y               =(2)
a2 + b1 + c0 + Z          =(3)
a3 + b2 + c1 + d0 + W     =(4)
     b3 + c2 + d1 + e0    =f0
          c3 + d2 + e1    =f1
               d3 + e2    =f2
                    e3    =f3
    (1)  (2)  (3)  (4)
```

Keterangan:
table [(1)] menghasilkan b3 b2 b1 b0
table [(2)] menghasilkan c3 c2 c1 c0
table [(3)] menghasilkan d3 d2 d1 d0
table [(4)] menghasilkan e3 e2 e1 e0
register awal : a3 a2 a1 a0
register akhir: f3 f2 f1 f0

Terlihat bahwa cara yang digunakan untuk reversing sama dengan cara yang kita gunakan pada CRC16.
Contoh penggunaan reversing CRC32 sebagai berikut:

- register awal : a3 a2 a1 a0 -> AB CD EF 66
- register akhir: f3 f2 f1 f0 -> 56 33 14 78
- setelah melakukan proses seperti contoh pada CRC16, kita dapatkan nilai (1),(2),(3),(4) dengan lebih dahulu mendapatkan byte awalnya, sebagai berikut:

Awal byte	entry	value
e3=f3	=56 -> 35h=(4)	56B3C423
d3=f2+e2	=E6 -> 4Fh=(3)	E6635C01
c3=f1+e1+d2	=B3 -> F8h=(2)	B3667A2E
b3=f0+e0+d1+c2	=61 -> DEh=(1)	616BFFD3

Maka nilai :
e3 e2 e1 e0 à 56B3C423
d3 d2 d1 d0 à E6635C01
c3 c2 c1 c0 à B3667A2E
b3 b2 b1 b0 à 616BFFD3

Sekarang kita telah mendapatkan semua variable yang diperlukan, maka nilai 4 byte baru dapat langsung kita hitung dengan rumusan sebagai berikut:

```
X=(1)+a0          =DE+66=B8
Y=(2)+b0+a1       =F8+D3+EF=C4
Z=(3)+c0+b1+a2    =4F+2E+FF+CD=53
W=(4)+d0+c1+b2+a3=35+01+7A+6B+AB
                   =8E
```

Kesimpulan dari penghitungan diatas, untuk mengubah isi register CRC32 dari **AB CD EF 66** menjadi **56 33 14 78** diperlukan 4 byte baru yaitu: **B8 C4 53 8E**

Berikut ini contoh kode dalam assembly yang digunakan untuk melakukan reverse, dalam assembly dwords ditulis dan dibaca dengan urutan terbalik.

```
crcBefore       dd (?)
wantedCrc       dd (?)
buffer          db 8 dup (?)

mov eax, dword ptr[crcBefore]
;/* Awal Step1
mov     dword ptr[buffer], eax
mov     eax, dword
ptr[wantedCrc]
mov     dword ptr[buffer+4], eax
; Akhir Step1*/
mov     di, 4

computeReverseLoop:
mov al, byte ptr[buffer+di+3]
;/* Awal Step1
call   GetTableEntry
; Step 2 */
xor dword ptr[buffer+di], eax
; Step 3
xor    byte ptr[buffer+di-1], bl
; Step 4
dec     di
;/*
jnz    computeReverseLoop
; Step 5 */
```

Notes:
-Registers eax, di bx are used

Implementation of GetTableEntry

```
crctable dd 256 dup (?)
;should be defined globally
;somewhere & initialized of
;course

mov  bx, offset crctable-1

getTableEntryLoop:
    add bx, 4
;points to (crctable-1)+k*4
;(k:1..256)
    cmp [bx], al
;must always find the value
;somewhere
    jne getTableEntryLoop

    sub     bx, 3
    mov     eax, [bx]
    sub     bx, offset crctable
    shr     bx, 2

    ret
```

eax berisi table entry, bx berisi entry number.

(ditulis oleh Anarchriz, dalam **http://www.woodmann.com/fravia/crctut1.htm)**

Untuk mempercepat dan menjadikan algoritma reverse lebih efektif, kita bisa menggunakan suatu *lookup reverse table* yang bisa kita simpan sebelumnya. Kita bisa membuat *lookup reverse table* dengan menyimpan semua byte awalan sebagai index yang mengacu pada nilai byte secara keseluruhan. Misalkan untuk nilai table 0500 (pada CRC16) kita menjadikan 05 sebagai index yang nilainya mengacu pada nilai 0500, yaitu table[05] = 0500. hal ini akan mempercepat perhitungan kita karena semua operasi pencarian dilakukan dengan sekali iterasi.

8. Aplikasi CRC

CRC merupakan salah satu jenis fungsi hash yang sering digunakan untuk menjaga integritas suatu data. Hal ini biasa diterapkan untuk proses transmisi data, storage, ataupun untuk aplikasi –aplikasi tertentu (misalnya *compressor* data).

Untuk menjaga integritas data, maka kita menyimpan nilai CRC dari suatu data bersamaan dengan data tersebut (baik dalam proses transmisi maupun storage). Autentikasi dilakukan dengan melakukan pengecekan terhadap nilai CRC yang dihitung dari data yang diterima dengan nilai CRC yang disimpan untuk data tersebut. Jika nilai CRC-nya sama, maka autentikasi berhasil (data tidak mengalami kerusakan selama proses transmisi ataupun storage).

Di sisi lain, suatu aplikasi mungkin hanya mengijinkan suatu data untuk memiliki nilai CRC tertentu. Hal ini cukup efektif karena selain digunakan untuk menjaga integritas data, juga bisa digunakan untuk memberi identitas file. Hal ini bisa kita lakukan dengan mudah, yaitu dengan menambahkan 4 byte data di akhir data sehingga keseluruhan data memiliki nilai CRC sesuai dengan nilai yang telah ditetapkan oleh programmer. Kita bisa melakukannya dengan memanfaatkan cara yang sama dalam melakukan reversing CRC, hanya kali ini kita meletakkan 4 byte baru di akhir file (asumsi kita menggunakan CRC32).

Dengan nilai CRC yang sudah ditentukan, kita masih bisa memodifikasi data dengan tetap mempertahankan nilai CRC yang sudah ada. Hal ini menjadi salah satu alas an digunakannya CRC untuk melakukan pengecekan terhadap integritas data. Tentu saja dengan keterbatasannya, integritas data yang dapat dijamin hanyalah integritas yang disebabkan perubahan beberapa bit yang tidak disengaja (karena alasan media penyimpanan, saluran pengiriman,dan lain-lain). Perubahan atau serangan terhadap data yang disengaja, akan sangat mudah untuk tetap dapat melewati uji CRC karena kita tetap bisa menjaga nilai CRC yang asli dengan data yang telah berubah, bahkan dengan perubahan yang sangat drastis.

CRC, jenis yang sering digunakan adalah CRC32, sudah diimplementasikan dalam berbagai bahasa pemrograman seperti java, C, dan php sebagai suatu fungsi yang sudah built in. Pengecekan nilai CRC juga dilakukan oleh aplikasi Winrar, yang setahu penulis digunakan apakah file yang dikompresi mengalami *corrupt file* atau tidak. Kerusakan (*corrupt*) ini bisa disebabkan oleh proses kompresi yang tidak berjalan sebagaimana mestinya ataupun karena proses penyimpanan maupun transmisi data. Kerusakan data diketahui dari nilai CRC ini akan mempermudah aplikasi sehingga tidak perlu menjalankannya (mengekstraknya) dulu baru diketahui adanya kerusakan file.

9. Analisis

Meskipun sebagai salah satu jenis fungsi hash, namun CRC tidak seaman fungsi hash lain yang biasa digunakan dalam dunia kriptografi, misalnya keluarga fungsi MD ataupun SHA.

CRC lebih efektif diterapkan pada level *hardware* karena bisa mempercepat operasinya, dan hal ini mudah dilakukan. Untuk menerapkannya pada level *hardware*, kita bisa menulis kode dalam bahasa assembly. Meskipun begitu, tidak masalah juga jika kita menggunakan bahasa pemrogramam (misalnya java ataupun C) untuk melakukan komputasi nilai CRC.

Perubahan satu bit saja dalam data akan merusak nilai CRC, sehingga kemampuan ini menjadikan CRC sesuai untuk melakukan pengecekan integritas data.

CRC juga mempunyai kemampuan untuk dilakukan *reverse*. Hal ini berarti, kita bisa melakukan modifikasi terhadap data yang kita miliki sehingga mempunyai nilai CRC tertentu. Kemampuan reverse ini mempunyai keuntungan bahwa kita tidak perlu merusak nilai CRC suatu data ketika kita ingin melakukan modifikasi data. Di sisi lain, kerugian yang ditimbulkan yaitu kita tidak bisa mengetahui kalo data yang kita dapat diubah dengan sengaja hanya dengan mengandalkan pengecekan nilai CRC.

10. Kesimpulan

Meskipun CRC tidak menjadi fungsi hash yang aman yang berguna untuk menjaga data dari perubahan disengaja yang tidak diinginkan, CRC masih bisa diandalkan untuk menjamin data yang kita dapatkan tidak mengalami kerusakan dalam proses transmisi dan *storage*-nya. Kemampuan *reverse* juga memudahkan bagi programmer untuk menjaga aplikasi yang dibuatnya hanya mengenali satu nilai CRC saja. Modifikasi data dapat dilakukan dengan tetap menjaga nilai asli CRC dengan menggunakan algoritma reverse CRC.

Jadi, CRC memang bukan dibuat untuk menggantikan fungsi hash semisal MD ataupun SHA, tetapi lebih berfungsi untuk melengkapi sehingga menjamin tidak ada kerusakan data yang tidak disengaja dengan tetap menjaga kesederhanaan algoritma. CRC masih bisa ditingkatkan kemampuannya, terutama dalam memilih bilangan polynomial (‘poly’) sehingga komputasi yang dihasilkan lebih efektif.

11. Lampiran

Untuk mempermudah dalam melakukan penghitungan secara manual, berikut ini penulis melampirkan table CRC untuk CRC16 dan CRC32.

CRC-16 Table

Idx	Nilai		Idx	Nilai
00h	0000		80h	A001
01h	C0C1		81h	60C0
02h	C181		82h	6180
03h	0140		83h	A141
04h	C301		84h	6300
05h	03C0		85h	A3C1
06h	0280		86h	A281
07h	C241		87h	6240
08h	C601		88h	6600
09h	06C0		89h	A6C1
0Ah	0780		8Ah	A781
0Bh	C741		8Bh	6740
0Ch	0500		8Ch	A501
0Dh	C5C1		8Dh	65C0
0Eh	C481		8Eh	6480
0Fh	0440		8Fh	A441
10h	CC01		90h	6C00
11h	0CC0		91h	ACC1
12h	0D80		92h	AD81
13h	CD41		93h	6D40
14h	0F00		94h	AF01
15h	CFC1		95h	6FC0
16h	CE81		96h	6E80
17h	0E40		97h	AE41
18h	0A00		98h	AA01
19h	CAC1		99h	6AC0
1Ah	CB81		9Ah	6B80
1Bh	0B40		9Bh	AB41
1Ch	C901		9Ch	6900
1Dh	09C0		9Dh	A9C1
1Eh	0880		9Eh	A881
1Fh	C841		9Fh	6840
20h	D801		A0h	7800
21h	18C0		A1h	B8C1
22h	1980		A2h	B981
23h	D941		A3h	7940
24h	1B00		A4h	BB01
25h	DBC1		A5h	7BC0
26h	DA81		A6h	7A80
27h	1A40		A7h	BA41
28h	1E00		A8h	BE01
29h	DEC1		A9h	7EC0
2Ah	DF81		AAh	7F80
2Bh	1F40		ABh	BF41
2Ch	DD01		ACh	7D00
2Dh	1DC0		ADh	BDC1
2Eh	1C80		AEh	BC81
2Fh	DC41		AFh	7C40
30h	1400		B0h	B401
31h	D4C1		B1h	74C0
32h	D581		B2h	7580
33h	1540		B3h	B541
34h	D701		B4h	7700
35h	17C0		B5h	B7C1
36h	1680		B6h	B681
37h	D641		B7h	7640
38h	D201		B8h	7200
39h	12C0		B9h	B2C1
3Ah	1380		BAh	B381
3Bh	D341		BBh	7340
3Ch	1100		BCh	B101
3Dh	D1C1		BDh	71C0
3Eh	D081		BEh	7080
3Fh	1040		BFh	B041
40h	F001		C0h	5000
41h	30C0		C1h	90C1
42h	3180		C2h	9181
43h	F141		C3h	5140
44h	3300		C4h	9301
45h	F3C1		C5h	53C0
46h	F281		C6h	5280
47h	3240		C7h	9241
48h	3600		C8h	9601
49h	F6C1		C9h	56C0
4Ah	F781		CAh	5780
4Bh	3740		CBh	9741
4Ch	F501		CCh	5500
4Dh	35C0		CDh	95C1
4Eh	3480		CEh	9481
4Fh	F441		CFh	5440
50h	3C00		D0h	9C01
51h	FCC1		D1h	5CC0
52h	FD81		D2h	5D80
53h	3D40		D3h	9D41
54h	FF01		D4h	5F00
55h	3FC0		D5h	9FC1
56h	3E80		D6h	9E81
57h	FE41		D7h	5E40
58h	FA01		D8h	5A00
59h	3AC0		D9h	9AC1
5Ah	3B80		DAh	9B81
5Bh	FB41		DBh	5B40
5Ch	3900		DCh	9901
5Dh	F9C1		DDh	59C0
5Eh	F881		DEh	5880
5Fh	3840		DFh	9841
60h	2800		E0h	8801

61h	E8C1		E1h	48C0
62h	E981		E2h	4980
63h	2940		E3h	8941
64h	EB01		E4h	4B00
65h	2BC0		E5h	8BC1
66h	2A80		E6h	8A81
67h	EA41		E7h	4A40
68h	EE01		E8h	4E00
69h	2EC0		E9h	8EC1
6Ah	2F80		EAh	8F81
6Bh	EF41		EBh	4F40
6Ch	2D00		ECh	8D01
6Dh	EDC1		EDh	4DC0
6Eh	EC81		EEh	4C80
6Fh	2C40		EFh	8C41
70h	E401		F0h	4400
71h	24C0		F1h	84C1
72h	2580		F2h	8581
73h	E541		F3h	4540
74h	2700		F4h	8701
75h	E7C1		F5h	47C0
76h	E681		F6h	4680
77h	2640		F7h	8641
78h	2200		F8h	8201
79h	E2C1		F9h	42C0
7Ah	E381		FAh	4380
7Bh	2340		FBh	8341
7Ch	E101		FCh	4100
7Dh	21C0		FDh	81C1
7Eh	2080		FEh	8081
7Fh	E041		FFh	4040

CRC-32 Table

Index Awal Nilai

Idx	Nilai		Idx	Nilai
00h	00000000		80h	EDB88320
01h	77073096		81h	9ABFB3B6
02h	EE0E612C		82h	03B6E20C
03h	990951BA		83h	74B1D29A
04h	076DC419		84h	EAD54739
05h	706AF48F		85h	9DD277AF
06h	E963A535		86h	04DB2615
07h	9E6495A3		87h	73DC1683
08h	0EDB8832		88h	E3630B12
09h	79DCB8A4		89h	94643B84
0Ah	E0D5E91E		8Ah	0D6D6A3E
0Bh	97D2D988		8Bh	7A6A5AA8
0Ch	09B64C2B		8Ch	E40ECF0B
0Dh	7EB17CBD		8Dh	9309FF9D
0Eh	E7B82D07		8Eh	0A00AE27
0Fh	90BF1D91		8Fh	7D079EB1

10h	1DB71064		90h	F00F9344
11h	6AB020F2		91h	8708A3D2
12h	F3B97148		92h	1E01F268
13h	84BE41DE		93h	6906C2FE
14h	1ADAD47D		94h	F762575D
15h	6DDDE4EB		95h	806567CB
16h	F4D4B551		96h	196C3671
17h	83D385C7		97h	6E6B06E7
18h	136C9856		98h	FED41B76
19h	646BA8C0		99h	89D32BE0
1Ah	FD62F97A		9Ah	10DA7A5A
1Bh	8A65C9EC		9Bh	67DD4ACC
1Ch	14015C4F		9Ch	F9B9DF6F
1Dh	63066CD9		9Dh	8EBEEFF9
1Eh	FA0F3D63		9Eh	17B7BE43
1Fh	8D080DF5		9Fh	60B08ED5
20h	3B6E20C8		A0h	D6D6A3E8
21h	4C69105E		A1h	A1D1937E
22h	D56041E4		A2h	38D8C2C4
23h	A2677172		A3h	4FDFF252
24h	3C03E4D1		A4h	D1BB67F1
25h	4B04D447		A5h	A6BC5767
26h	D20D85FD		A6h	3FB506DD
27h	A50AB56B		A7h	48B2364B
28h	35B5A8FA		A8h	D80D2BDA
29h	42B2986C		A9h	AF0A1B4C
2Ah	DBBBC9D6		AAh	36034AF6
2Bh	ACBCF940		ABh	41047A60
2Ch	32D86CE3		ACh	DF60EFC3
2Dh	45DF5C75		ADh	A867DF55
2Eh	DCD60DCF		AEh	316E8EEF
2Fh	ABD13D59		AFh	4669BE79
30h	26D930AC		B0h	CB61B38C
31h	51DE003A		B1h	BC66831A
32h	C8D75180		B2h	256FD2A0
33h	BFD06116		B3h	5268E236
34h	21B4F4B5		B4h	CC0C7795
35h	56B3C423		B5h	BB0B4703
36h	CFBA9599		B6h	220216B9
37h	B8BDA50F		B7h	5505262F
38h	2802B89E		B8h	C5BA3BBE
39h	5F058808		B9h	B2BD0B28
3Ah	C60CD9B2		BAh	2BB45A92
3Bh	B10BE924		BBh	5CB36A04
3Ch	2F6F7C87		BCh	C2D7FFA7
3Dh	58684C11		BDh	B5D0CF31
3Eh	C1611DAB		BEh	2CD99E8B
3Fh	B6662D3D		BFh	5BDEAE1D
40h	76DC4190		C0h	9B64C2B0
41h	01DB7106		C1h	EC63F226
42h	98D220BC		C2h	756AA39C
43h	EFD5102A		C3h	026D930A
44h	71B18589		C4h	9C0906A9
45h	06B6B51F		C5h	EB0E363F

46h	9FBFE4A5		C6h	72076785
47h	E8B8D433		C7h	05005713
48h	7807C9A2		C8h	95BF4A82
49h	0F00F934		C9h	E2B87A14
4Ah	9609A88E		CAh	7BB12BAE
4Bh	E10E9818		CBh	0CB61B38
4Ch	7F6A0DBB		CCh	92D28E9B
4Dh	086D3D2D		CDh	E5D5BE0D
4Eh	91646C97		CEh	7CDCEFB7
4Fh	E6635C01		CFh	0BDBDF21
50h	6B6B51F4		D0h	86D3D2D4
51h	1C6C6162		D1h	F1D4E242
52h	856530D8		D2h	68DDB3F8
53h	F262004E		D3h	1FDA836E
54h	6C0695ED		D4h	81BE16CD
55h	1B01A57B		D5h	F6B9265B
56h	8208F4C1		D6h	6FB077E1
57h	F50FC457		D7h	18B74777
58h	65B0D9C6		D8h	88085AE6
59h	12B7E950		D9h	FF0F6A70
5Ah	8BBEB8EA		DAh	66063BCA
5Bh	FCB9887C		DBh	11010B5C
5Ch	62DD1DDF		DCh	8F659EFF
5Dh	15DA2D49		DDh	F862AE69
5Eh	8CD37CF3		DEh	616BFFD3
5Fh	FBD44C65		DFh	166CCF45
60h	4DB26158		E0h	A00AE278
61h	3AB551CE		E1h	D70DD2EE
62h	A3BC0074		E2h	4E048354
63h	D4BB30E2		E3h	3903B3C2
64h	4ADFA541		E4h	A7672661
65h	3DD895D7		E5h	D06016F7
66h	A4D1C46D		E6h	4969474D
67h	D3D6F4FB		E7h	3E6E77DB

68h	4369E96A		E8h	AED16A4A
69h	346ED9FC		E9h	D9D65ADC
6Ah	AD678846		EAh	40DF0B66
6Bh	DA60B8D0		EBh	37D83BF0
6Ch	44042D73		ECh	A9BCAE53
6Dh	33031DE5		EDh	DEBB9EC5
6Eh	AA0A4C5F		EEh	47B2CF7F
6Fh	DD0D7CC9		EFh	30B5FFE9
70h	5005713C		F0h	BDBDF21C
71h	270241AA		F1h	CABAC28A
72h	BE0B1010		F2h	53B39330
73h	C90C2086		F3h	24B4A3A6
74h	5768B525		F4h	BAD03605
75h	206F85B3		F5h	CDD70693
76h	B966D409		F6h	54DE5729
77h	CE61E49F		F7h	23D967BF
78h	5EDEF90E		F8h	B3667A2E
79h	29D9C998		F9h	C4614AB8
7Ah	B0D09822		FAh	5D681B02
7Bh	C7D7A8B4		FBh	2A6F2B94
7Ch	59B33D17		FCh	B40BBE37
7Dh	2EB40D81		FDh	C30C8EA1
7Eh	B7BD5C3B		FEh	5A05DF1B
7Fh	C0BA6CAD		FFh	2D02EF8D

Keterangan:

Idx menyatakan index table dari nilai table. Cara membaca table adalah 2 kolom pertama dari atas sampai ke bawah baru kemudian 2 kolom terakhir. 2 kolom pertama berisi 128 nilai pertama dari table, dan 2 kolom terakhir berisi 128 nilai selanjutnya.

12. Daftar Pustaka

http://www.wikipedia.org/ Crc32

http://www.informatika.org/~rinaldi

Munir, Rinaldi. 2006. Diktat Kuliah IF5054 kriptografi. Bandung.

http://www.woodmann.com/fravia/c rctut1.htm

http://sar.informatik.hu-berlin.de/research/publications/

http://www.cs.cmu.edu/%7edst/CP4break/cp4break.html

http://www.repairfaq.org/filipg/LINK/F%5fcrc%5fv3.html

Stigge,Martin. May, 2006. Reversing CRC, theory and practice.

Peterson,W.W. and Brown, D.T. “Cyclic Codes for Error Detection.” In *Proceedings of the IRE*, January 1961, 228–235.

Kowalk. August, 2006. CRC Cyclic Redundancy Check Analysing and Correcting Errors.