UNIVERZA V LJUBLJANI
Fakulteta za računalništvo in informatiko

KONTROLNE VSOTE (Checksums)

CRC32, CRC16, CRC16 CITT,...

Tečaj iz kriptografije in računalniške varnosti
Predavatelj: Aleksandar Jurišič

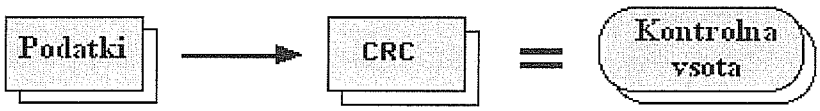
Robert Kuster

Ljubljana, 2002

0. Kazalo

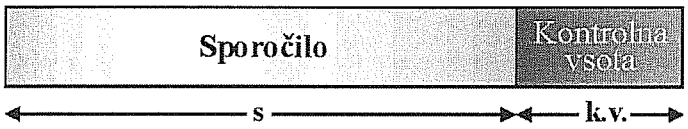
1. Uvod – Kaj je CRC?

CRC algoritme uvrščamo med takoimenovane zgoščevalne funkcij in so ključna komponenta za odkrivanje napak (Error Detection) v mnogih sistemih. CRC algoritem izvedemo nad nekim blokom podatkov z namenom, da dobimo število, ki nam enolično predstavlja tako vsebino kot tudi samo organizacijo podatkov. CRC algoritem si torej lahko predsavljamo kot funkcijo, ki generira »prstni odtis« nekega bloka podatkov. »Prstni odtis« oz. dobljeno število imenujemo **kontrolna vsota**.



Slika 1: Shematičen prikaz CRC preračuna

Če primerjamo kontrolno vsoto bloka podatkov s kontrolno vsoto drugega bloka podatkov, lahko ugotovimo, če sta bloka identična ali ne. CRC algoritme uporablja za preverjanje pravilnosti podatkov tudi večina mrežnih protokolov. Pošiljatelj v ta namen izračuna kontrolno vsoto in jo pripne originalnemu sporočilu. Tudi prejemnik izračuna kontrolno vsoto z isto funkcijo in jo primerja s pripeto kontrolno vsoto.



Slika 2: Sporočilo s pripeto kontrolno vsoto

2. Potreba po kompleksnosti

Kot primer vzemimo kontrolno funkcijo, ki zgolj sešteje bajte v sporočilu (po modulu 256):

Sporočilo:	6	23	4	
Sporočilo s kontrolno vsoto:	6	23	4	33
Spremenjeno sporočilo:	6	27	4	33

V zgornjem primeru se je spremenila vsebina drugemu bajtu sporočila, vendar bo prejemnik sedaj napako lahko zaznal. Kljub temu, da nam že sorazmerno preprosta operacija kot je seštevanje omogoča odkritje določene vrste napak, je taka funkcija načeloma preveč primitivna. Pri takem algoritmu namreč vsak vhodni bajt sporočila načeloma vpliva le na en bajt kontrolne voste. Napaka v sledečem primeru bi tako ostala neodkrita:

Sporočilo:	6	23	4	
Sporočilo s kontrolno vsoto:	6	23	4	33
Spremenjeno sporočilo:	8	20	5	33

Problem lahko rešimo le z uporabo bolj dovršene formule, pri kateri bi vsak vhodni bit sporočila vplival na vsak bit kontrolne vsote, ne le na en bajt.

Zgornji algoritem pa ima še eno pomankljivost. Širina konrolnega registra (hrani vrednost kontrolne vsote) je le en bajt – za kontrolno vsoto imamo tako na razpolago le 256 različnih vrednosti, kar z drugimi besedami pomeni sledeče: ob pojavu naključne napake obstaja verjetnost 1:256, da le te sploh ne odkrijemo.

Vidimo, da za dober algoritem moramo upoštevati vsaj dvoje:
- širina kontrolnega registra naj bo dovolj velika, da je verjetnost neodkritih napak (zaradi kolosij) dovolj majhna
- vsak vhodni bit sporočila naj ima možnost, da vpliva na poljubno število bitov v kontrolnem registru oz. kontrolni vsoti

Izraz kontrolna vsota se je včasih uporabljal za opis enostavnejših algoritmov, baziranih na seštevanju. Danes je njegov pomen bolj splošen in se uporablja predvsem v povezavi z bolj zapletenimi algoritmi, kot je npr. CRC.

3. Osnovna ideja CRC algoritmov

Medtem ko seštevanje očitno ni dovolj močna operacija za tvorbo kontrolne vsote se izkaže, da deljenje je. Pogoj je le, da je delitelj vsaj tako dolg kot je dolžina kontrolnega registra.

Osnovna ideja CRC algoritmov je:
Sporočilo obravnavamo kot veliko binarno število, ki ga delimo z drugim fiksnim številom, ostanek pa je kontrolno vsota.

$$CRC\ vrednost = ostanek\ od\ \frac{sporočilo}{x^{16} + x^{12} + x^5 + 1}$$

4. Polinomska aritmetika

Pri obravnavanju CRC algoritmov pogosto naletimo na besedo »polinom«. Namesto da bi delitelj, deljenec (sporočilo), kvocient in ostanek obravnavali kot naravna števila, gledamo na njih kot na polinome z binarnimi koeficienti. V ta namen si število predstavljamo kot binarni niz, kjer so vrednosti bitov koeficienti polinoma.
Za primer si poglejmo število 23 (dec), kar je 17 (hex) oz. 1011 v binarnem zapisu. Koeficienti polinoma so tako $1 \cdot x^4 + 0 \cdot x^3 + 1 \cdot x^2 + 1 \cdot x^1 + 1 \cdot x^0$ ali krajše $x^4 + x^2 + x^1 + x^0$.

Z uporabo te tehnike sporočilo (deljenec) in delitelj (generatorski polinom) predstavimo s polinomi. Izvajamo lahko vse aritmetične opracije, ki smo jih tudi s števili – če npr. hočemo zmnožiti 1101 in 1011, to storimo tako:

$$(x^3 + x^2 + x^0) \cdot (x^3 + x^1 + x^0) =$$
$$(x^6 + x^4 + x^3 +$$
$$x^5 + x^3 + x^2 +$$
$$x^3 + x^1 + x^0) = x^6 + x^5 + x^4 + 3 \cdot x^3 + x^2 + x^1 + x^0$$

V polinomski aritmetiki ne poznamo povezave med koeficienti, če ne vemo koliko je x (ne vemo npr., da je $3 \cdot x^3$ enako kot $x^4 + x^3$, če ne vemo, da je $x = 2$). Tako poznamo več polinomskih aritmetik, ki se med seboj razlikujejo zgolj po tem, kako so posamezni koeficienti med seboj povezani. Za nas je še posebej zanimiva polinomska aritmetika, pri kateri so koeficienti izračunani brez prenosov in po modulu 2. To je takoimenovana »polinomska aritmetika po modulu 2«. Vsi koeficienti so 0 ali 1.

Za naš primer tako sledi:

$$(x^3+x^2+x^0)\cdot(x^3+x^1+x^0)=$$
$$(x^6+x^4+x^3+$$
$$x^5+x^3+x^2+$$
$$x^3+x^1+x^0)=x^6+x^5+x^4+3\cdot x^3+x^2+x^1+x^0$$
$$=x^6+x^5+x^4+x^3+x^2+x^1+x^0$$

Očitno je, da je polinomska aritmetika po modulu 2 ekvivalentna binarni aritmetiki brez prenosov.

5. Binarna aritmetika brez prenosov ali CRC aritmetika

Vse računske operacije povezane s CRC algoritmi so izvršene v binarni aritmetiki brez prenosov, zato jo včasih imenujemo kar CRC aritmetika. Kot primer si poglejmo seštevanje dveh števil v CRC aritmetiki:

```
 1011
+1010
------
 0001
```

Odštevanja je ekvivalentno seštevanju. V bistvu sta v CRC aritmetiki tako seštevanje kot odštevanje ekvivalentna XOR operaciji, ki je inverz zamega sebe.

Za vsak bit obstajajo tako le 4 možnosti:

	0	0	1	1
+,-,XOR	0	1	0	1
	0	1	1	0

Z definiranim seštevanjem lahko definiramo tudi množenje in deljenje. Množenje je povsem očitno:

```
  1101
*1011
------
  1101
 0000
1101
1101
------
1111111
-------
```

Nekoliko bolj zapleteno je deljenje, saj moram vedeti kdaj je neko število deljivo z drugim. Glede na to, da sta v CRC aritmetiki seštevanje in odštevanje ekvivalentna, je pojem velikosti števil zabrisan. Medtem ko je očitno, da je število 1010 večje kot 10, pa to ne velja za števili 1010 in 1001 (k številu 1010 lahko bodisi prištejemo ali odštejemo število 0011; v obeh primerih dobimo 1001). Tako postane običajno pojmovanje velikosti nesmiselno, zato pojem velikosti najprej definiramo:

X je večji ali kvečjemu enak Y, če je pozicija najvišjega bita (z vrednostjo 1) v X na enakem ali večjem (=bolj levem) mestu kot pozicija tega bita v Y.

Deljenje potem izgleda tako:

```
 1101110 / 1011=1
1011
 ▒101
  1011
  ▒101
   1011
   ▒100
    1011
      111 = ostanek
```

V splošnem velja v CRC aritmetiki dejstvo: če je število A mnogokratnik števila B to pomeni, da je možno A konstruirati tako, da vrednost 0 večkrat XORamo (pri različnih zamikih) s številom B.

6. Primer CRC algoritma

Za izračun CRC vrednosti najprej izberemo delitelj, imenovan »generatorski polinom«. Le ta je ključna komponenta vseh CRC algoritmov.
Širina polinoma W je zelo pomembna in določa celoten preračun. Danes so v uporabi večinoma polinomi širine 16 in 32, ker se tako poenostavi implementacija samega algoritma na sodobnih računalnikih. Širina polinom je v bistvu pozicija najvišjega bita, ki ima vrednost 1 – širina polinoma 10011 je tako 4 in ne 5.

Ko izberemo polinom, nadaljujemo z iračunom: sporočilo delimo (v CRC aritmetiki) z izbranim polinomom; edina posebnost je, da sporočilo pred deljenjem podaljšamo za W ničel.

Izvorno sporočilo:	1101011011
Polinom:	10011
Sporočilo podaljšano za W ničel:	11010110110000

Podaljšano sporočilo nato delimo z izbranim polinomom. Za naš primer bi dobili kvocient 1100001010 (nas ne zanima) in ostanek 1110, ki predstavlja kontrolno vsoto. Kontrolno vsoto običajno pripnemo sporočilu, ki ga nato odpošljemo. V našem primeru torej dobimo vrednost: 11010110111110.

Prejemnik sporočila lahko naredi sledeče:

1. Loči sporočilo in kontrolno vsoto. Izračuna kontrolno vsoto sporočila (najprej doda W ničel) in obe kontrolni vsoti primerja.
2. Izračuna kontrolno vsoto vsega kar prejme (brez dodajanja ničel). Rezultat bi moral biti 0.

Oba načina sta ekvivalentna.

Povzetek operacij pri CRC algoritmih:

1. Izberemo generatorsku polinom G s širino W
2. Sporočilu pripnemo W ničel. Razširjeno sporočilo naj bo S'.
3. S' delimo z G (v CRC aritmetiki). Ostanek je kontrolna vsota.

7. Izbira polinoma

Pri izbiri polinoma moramo biti previdni, saj so nekateri boljši od drugih. Najbolje je, če izberemo kar kakega standardnega, ki je že testiran. Po drugi strani pa si lahko izmislimo tudi čisto nov polinom – tako dobimo svoj CRC algoritem, pri čemer pa moramo upoštevati nekaj dejstev.

Najprej je potrebno vedeti, da je poslano sporočilo T mnogokratnik generatorskega polinoma G. Zadnjih W bitov sporočila T je namreč ostanek, ki ostane pri deljenju podaljšanega sporočila (za W ničel) s polinomom G. Če se sporočilo med prenosom poškoduje, bomo namesto sporočila T prejeli T+E, kjer je E vektor napake. Prejemnik sporočila bo tako delil T+E s polinomom G. Ker je $T \bmod G = 0 \quad \rightarrow \quad (T+E) \bmod G = E \bmod G$. Iz tega vidimo, da bo vsaka napaka, ki bo mnogokrantik izbranega polinoma G, ostala neodkrita. Izbrati moramo torej polinom, katerega mnogokratniki bodo čim manj podobni šumu in motnjam, ki se pojavljajo tekom prenosnega medija.

V splošnem poznamo sledeče vrste napak:

1.) enobitne napake (E=1000...00000) *
   Te vrste napak odkrijemo vedno, če ima polinom G vsaj dva bita z vrednostjo 1.
2.) Dvobitne napake (E=100...000100...000) *
   Da bi odkrili vse napake te vrste, moramo izbrati polinom, ki nima mnogokgatnikov tipa 11, 101, 1001, 10001, 100001,....
3.) Napake z lihim številom bitov. *
   Z izbiro polinoma, ki ima sodo število bitov z vrednostjo 1, odkrijemo vse napake, ki imajo liho število bitov z vrednostjo 1. Večina CRC algoritmov uporablja polinome s sodim številom bitvo z vrednostjo 1.
4.) Grozdne napake (E=00...00111..1111000...00) *
   V primeru da je širina grozdne napake E manjša od širine polinoma W, bo E vedno nedeljiv z izbranim polinomom → tako napako bomo gotovo odkrili. Če pa je širina grozdne napake večja od W, je verjetnost da je ne odkrijemo $(1/2)^W$.

Najvišji bit pri generatorskem polinomu se vedno podaja implicitno. Če je za nek CRC algoritem rečeno da uporablja polinom 0x1(hex)= 0001(bin), to pomeni, da je dejanski polinom enak 10001.

Zanimivo je tudi sledeče dejstvo: če je nek polinom dober, dobimo dober polinom tudi, če le tega prezrcalimo → če je 1011 dober polinom, je dober tudi polinom 1101. Ko kaka organizacija (npr. CITT) standarizira nek dober polinom, se tako prej ali slej začne uporabljati tudi njegova zrcaljena oblika, npr:

X25 standard: 1-0001-0000-0010-0001
X25 reversed: 1-0000-1000-0001-0001

CRC 16 standard: 1-1000-0000-0000-0101
CRC 16 reversed: 1-0100-0000-0000-0011

* *glej ref. [6] za podrobnejšo razlago*

Nekateri standardni polinomi:

CRC-12: $x^{12}+x^{11}+x^3+x^2+x+1$

X25 CRC: $x^{16}+x^{12}+x^2+1$

CRC-16: $x^{16}+x^{15}+x^2+1$

CRC-CITT: $x^{16}+x^{12}+x^5+1$

CRC-32: $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$

8. Realizacija CRC algoritma

Za realizacijo CRC algoritma moramo na nek način implementirati deljenje. Obstajata dva razloga, da ne moremo uporabiti kar instrukcije deljenja, prisotne na večini računalnikov:

1. deliti moramo v CRC aritmetiki;
2. sporočilo je lahko dolgo tudi več megabajtov, današnji računalniki pa nimajo tako velikih registrov;

Zato za implementacijo CRC deljenja pošljemo sporočilo (predstavljamo si ga kot tok bitov) skozi takoimenovani delilni register. S pomočjo delilnega registra lahko deljenje (v CRC aritmetiki) izvedemo enostavno s pomiki in XOR operacijami.

Za primer vzemimo polinom 10111. Ker je njegova širina 4, potrebujemo 4-bitni delilni register:

	3	2	1	0	biti registra
izhodni bit ←	?	?	?	?	← sporočilo
1	0	1	1	1	=polinom

Slika 3: Deljenje z delilnim registrom

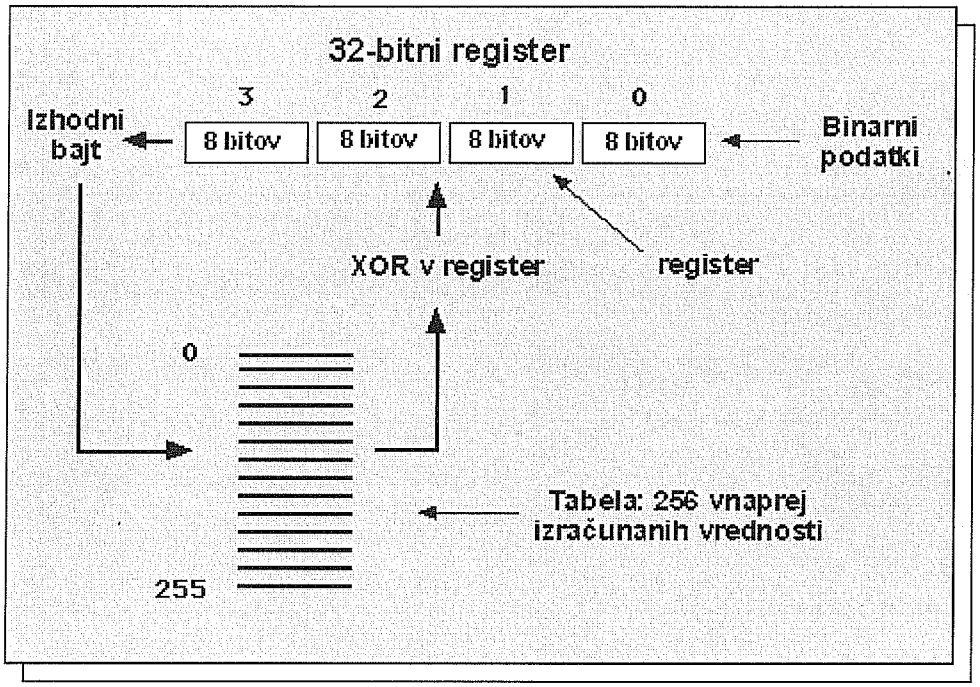
Deljenje izvedemo po sledečem pravilu:

1. vse bite v registru postavimo na vrednost 0
2. izvornemu sporočilu pripnemo W bitov z vrednostjo 0
3. premaknemo vrednosti registra za en bit v levo, na pozicijo 0 pa postavimo naslednji bit sporočila
4. če je vrednost izhodnega bita 1 → register=register XOR polinom
5. ponavljamo koraka 3 in 4, dokler sporočila ne zmanjka
6. v registru je vrednost ostanka

Kot vidimo je bil algoritem zasnovan tako, da ga je zelo enostavno (strojno) realizirati - s samimi XOR operacijami in zamiki.

9. CRC izvedba s tabelo

Medtem ko se da pravkar predstavljeni algoritem zelo elgantno implementirat na strojnem nivoju, pa je v primeru programske realizacije precej okoren in počasen pri izvajanju, saj deluje na bitnem nivoju. Da bi ga pohitrili potrebujemo način, s katerim bi obdelali več kot 1 bit sporočila naenkrat, npr. 4, 8, 16 ali 32 bitov. V nadaljevanju se bomo omejili na pomike po 8 bitov (1 bajt), ki jih uporablja večina današnjih (programsko realiziranih) CRC algoritmov. Pri tem si pomagamo s sledečim dejstvom: če neko konstantno (večkrat) XORamo v register pri različnih zamikih, potem obstaja neka **vrednost**, ki bo imela, če jo XORamo v register, enak efekt kot vse ostale XOR operacije. Te **vrednosti** lahko izračunamo in tabeliramo že vnaprej. Če pomikamo sporočilo v korakih po 1 bajt, bo v dobljeni tabeli 256 vrednosti (pri vsakem pomiku dobimo 1 izodni bajt, kar pomeni 256 različnih možnosti).



Slika 4: CRC izvedba s tabelo

Tako dobimo sledeč algoritem za izračun vrednosti CRC:

```
While (augmented message is not exhaused)
   Begin
   Top = top_byte(Register);
   Register = (Register << 24) | next_augmessage_byte;
   Register = Register XOR precomputed_table[Top];
   End
```

Zgornji algoritem je zelo učinkovit: za 1 bajt sporočila potrebujemo le en zamik (shift), eno ALI operacijo, eno XOR operacijo ter eno branje iz tabele.

10. Zrcalna izvedba

Definicija:
Neka vrednost/register je zrcaljena, če so njeni biti prezrcaljeni okrog sredine. 0101 je na primer 4 bitna zrcalna vrednost od 1010.

UART čipi (služijo serijeksemu V/I – »serial I/O«) delujejo tako, da vsak bajt odpošljejo z najmanj signifikantnim bitom (bit 0) spredaj, in najbolj signifikantnim bitom (bit 7) na koncu – torej vsak bajt pošljejo v zrcaljeni obliki. Posledica tega je, da so inženirji, ki so konstruirali CRC algoritme strojnem nivoju, dejansko naredili algoritme, kjer je bil vsak bajt sporočila prezrcaljen. Bajti so sicer bili procesirani v enakem vrstnem redu, toda biti znotraj vsakega bajta so bili prezrcaljeni: bit 0 v 7, bit 1 v 6, itd...
Zrcaljena izvedba se je kasneje pojavila tudi v »softwareski« obliki. Tako danes približno polovica algoritmov v praksi uporablja zrcaljeno obliko, čeprav le ta nima nobene prednosti pred običajno izvedbo.

11. Začetna in končna vrednost

CRC algoritmi se med seboj razlikujejo še v dveh lasnostih:
- začetna vrednost registra
- vrednost, ki jo XORamo k dobljenemu rezultatu

Pri CRC-32 algoritemu npr. inicializiramo delilni register z vrednostjo 0xFFFFFFFF, h končnemu rezultatu pa še XORamo vrednost 0xFFFFFFFF. Večina ostalih algoritmov začetno vrednost registra inicializira na 0.
Če predpostavimo, da so vse vrste sporočil in napak enako verjetne, nima začetna vrednost registra nobenega vpliva na to, kako dober je CRC algoritem. Ker pa se v praksi pojavljajo nekatere vrste sporočil pogosteje kot druge je dobro, če register incializiramo z vrednostjo, ki nima t.i. šibkih točk. Če npr. register incializiramo z 0, tak CRC algoritem ne bo zaznal razlike med dvema sporočiloma, ki se bosta razlikovali le po številu ničel na samem začetku. Ker pa so sporočila, ki se začnejo s samimi ničlami pogosta, je dobro, če register inicializiramo z vrednostjo različno od nič.

12. Parametri CRC-32, CRC-16 in CRC16-CITT algoritmov

Tabela 1: Parametri CRC algoritmov za vlad "123456789"

Ime	CRC16-CITT	CRC-16	CRC-32
Širina	16	16	32
Polinom	0x1021	0x8005	0x04C11DB7
Začetna vrednost registra	0xFFFF	0x0000	0xFFFFFFFF
Končni XOR	0x0000	0x0000	0xFFFFFFFF
Zrcaljeni podatki	Ne	Da	Da
Kontrolna vrednost	0x29B1	0xBB3D	0xCBF43296

13. Uporaba v kriptografiji in praksi

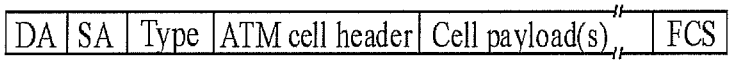
Kot smo omenili že na začetku, spadajo CRC algoritmi v skupino t.i. zgoščevalnih funkcij. Zgoščvelane funkcije se v kriptografiji uporabljajo za več stvari, največji pomen pa imajo v povezavi z digitalnimi podpisi;

> Ker so zgoščevalne funkcije načeloma hitrejše kot algoritmi za digitalne podpise, se digitalni podpis velikokrat izračuna le za zgostitev dokumenta, ki je veliko manjša, kot celoten dokument.

Zgostitev lahko tudi objavimo, brez strahu, da bi razkrili vsebino originalnega sporočila

Žal pa se izkaže, da je 32 bitna zgostitev (npr. CRC-32) za kriptografijo neuporabna. Pri napadu s paradoksom rojstnih dnevov bi namreč že z 2^{16} naključnimi čistopisi prišli do trčenja vsaj z verjetnostjo ½ (glej ref. [7], str. 94). Za uporabo v kriptografiji se zato priporoča vsaj 128-bitna zgostitev, ki jo dobimo pri MD2, MD4 ali MD5 algoritmih.

Drugače je v »nekriptografski« praksi. CRC algoritmi se namreč uporabljajo v mnogih komunikacijskih protokolih za odkrivanje napak – ravno to pa je naloga, za katero so bili pravzaprav tudi razviti. 32 bitni CRC algoritem (CRC-32) se npr. uporablja za odrivanje napak v IEEE 802 LAN omrežjih. Primer CIF (Cells in Frames) Ethernet paketa je prikazan na sliki 5.



Slika 5: Ethernet paket

DA = Ethernet Destination Address
SA = Ethernet Source Address
Type= Ethernet type field
FCS = Etherent Frame Check Sequence (CRC-32)

Kot vidim, je CRC-32 vrednost enostavno dodana na konec sporočila. CRC-32 uporabljata (za odkrivanje napak) tudi kompresijska programa WinZip in PKZIP.
Tudi pri ostalih mrežnih protokolih je uporaba podobna, s tem da nekateri uporabljajo drugačne CRC algoritme (t.j. polinome). Zanimivo je, da se za izračun kontrolnih vsot včasih uporabljajo še enostavnejši načini - za izračun kontrolne vsote zaglavja v IP paketu je npr. uporabljena zgolj operacija seštevanja. Iz tega lahko zaključimo, da CRC algortimi povsem zadostujejo potrebam za katere so bili razviti – to je odkrivanju napak v mrežnih protokolih.

14. Reference

[1] http://www.4d.com/ACIDOC/CMU/CMU79909.HTM
[2] http://www.repairfaq.org/filipg/LINK/F_crc_v3.html

[3] http://www.cee.hw.ac.uk/~pjbk/nets/crctutorial.html
[4] http://www.embedded.com/internet/9912/9912ia1.htm
[5] http://www.embedded.com/internet/0001/0001connect.htm

[6] Andrew S. Tanenbaum, **Computer Networks**, Prentice-Hall International Editions
[7] RSA Laboratories, **Frequently Asked Questions About Todays Cryptographiy**, 1996 RSA Data Security

15. Priloga

```
class CCRC
{
protected:
   CCRC();      // constructor

   ULONG crc32_table[256];            // Lookup table array
   WORD  crc16_table[256];            // Lookup table array
   WORD  crc16citt_table[256];        // Lookup table array

   void Init_CRC32_Table();           // Builds a Lookup table array
   void Init_CRC16_Table();
   void Init_CRC16CITT_Table();

   ULONG Reflect(ULONG, char);        // Reflects data bits

public:
   int  Get_CRC32 (CString&, DWORD);  // Calculates CRC-32 value from a string buffer
   WORD Get_CRC16 (CString&, DWORD);  // Calculates CRC-16 value from a string buffer
   WORD Get_CRC16CITT (CString&, DWORD);  // Calculates CRC-16CITT value from a string buffer
};


////////////////////////////////////////////////////////////////////////////////


CCRC::CCRC()    // constructor
{
   // Create lookup tables first
   Init_CRC32_Table();
   Init_CRC16_Table();
   Init_CRC16CITT_Table();
}
```

```
////////////////////////////////////////////////////////////////////////////////
//
// Initialize CRC tables.
//

void CCRC::Init_CRC32_Table()
{
   // The CRC32 polynomial.
   ULONG ulPolynomial = 0x04c11db7;

   // 256 values representing ASCII character codes.
   for(int i = 0; i <= 0xFF; i++)
   {
              crc32_table[i]=Reflect(i, 8) << 24;
              for (int j = 0; j < 8; j++)
              crc32_table[i] = (crc32_table[i] << 1) ^ (crc32_table[i] & (1 << 31) ? \
                                                                    ulPolynomial : 0);
              crc32_table[i] = Reflect(crc32_table[i], 32);
   }
}

void CCRC::Init_CRC16_Table()
{
   // The CRC16 polynomial.
   WORD wPolynomial = 0x8005;

   // 256 values representing ASCII character codes.
   for(int i = 0; i <= 0xFF; i++)
   {
              crc16_table[i]=Reflect(i, 8) << 8;
              for (int j = 0; j < 8; j++)
              crc16_table[i] = (crc16_table[i] << 1) ^ (crc16_table[i] & (1 << 15) ? \
                                                                    wPolynomial : 0);

              crc16_table[i] = Reflect(crc16_table[i], 16);
   }
}

void CCRC::Init_CRC16CITT_Table()
{
   // The CRC16 polynomial.
   WORD wPolynomial = 0x1021;


   // 256 values representing ASCII character codes.
   for(int i = 0; i <= 0xFF; i++)
   {
              crc16citt_table[i] = i << 8;
              for (int j = 0; j < 8; j++)
              crc16citt_table[i] = (crc16citt_table[i] << 1) ^ (crc16citt_table[i] & \
                                                        (1 << 15) ? wPolynomial : 0);
   }
}


////////////////////////////////////////////////////////////////////////////////


ULONG CCRC::Reflect(ULONG ref, char ch)
{
   ULONG value(0);

   // Swap bit 0 for bit 7
   // bit 1 for bit 6, etc.
   for(int i = 1; i < (ch + 1); i++)
   {
              if(ref & 1)
              value |= 1 << (ch - i);
              ref >>= 1;
   }
   return value;
}
```

```
////////////////////////////////////////////////////////////////////////////////
//
// Calculate CRC values
//

int CCRC::Get_CRC32 (CString &csData, DWORD dwSize)
{
   // initial value = 0xffffffff
   ULONG  crc(0xffffffff);
   int len = dwSize;
   unsigned char* buffer = (unsigned char*)(LPCTSTR)csData;

   // Perform the CRC32 algorithm on each character
   // in the buffer.
   while(len--)
       crc = (crc >> 8) ^ crc32_table[(crc & 0xFF) ^ *buffer++];

   // Final XOR
   return crc^0xffffffff;
}


WORD CCRC::Get_CRC16 (CString &csData, DWORD dwSize)
{
   // initial value = 0x0000
   WORD  crc(0x0000);
   int len = dwSize;
   unsigned char* buffer = (unsigned char*)(LPCTSTR)csData;

   // Perform the CRC16 algorithm on each character
   // in the buffer.
   while(len--)
       crc = (crc >> 8) ^ crc16_table[(crc & 0xFF) ^ *buffer++];

   return crc;
}


WORD CCRC::Get_CRC16CITT (CString &csData, DWORD dwSize)
{
   // initial value = 0xffff
   WORD  crc(0xffff);
   int len = dwSize;
   unsigned char* buffer = (unsigned char*)(LPCTSTR)csData;

   // Perform the CRC16CITT algorithm on each character
   // in the buffer.
   while(len--)
       crc = (crc << 8) ^ crc16citt_table[((crc >> 8) & 0xFF) ^ *buffer++];

   return crc;
}
```

Opomba:
Pravilnost delovanja sem preveril s kontrolnimi vrednostmi (glej tabelo na strani 8) - kot vhod vzamemo niz »123456789«.