Studia i Materiały Informatyki Stosowanej, Tom 6, Nr 14, 2014
str. 11-17

Studia i Materiały Informatyki Stosowanej, Tom 6, Nr 14, 2014
str. 11-17

IMPLEMENTACJA ALGORYTMU FUNKCJI SKRÓTU MD5 W JĘZYKU C++

Wojciech Makowski

Uniwersytet Kazimierza Wielkiego
Wydział Matematyki, Fizyki i Techniki
Instytut Techniki
e-mail: wojmak9@gmail.com

Streszczenie: *Algorytm funkcji skrótu MD5 to jeden z najpopularniejszych sposobów uzyskania skrótu wiadomości. Otrzymane skróty mogą służyć jako podpisy cyfrowe plików lub ciągów znaków. Niniejszy artykuł przedstawia implementację tego algorytmu w języku C++. Można tu znaleźć opis klasy, która może później posłużyć jako biblioteka do dowolnego programu napisanego w tym języku.*

Słowa kluczowe: *Kryptografia, Funkcja skrótu, podpis cyfrowy, C++*

Implementation of Message-Digest algorithm 5 in C++

Abstarct: *Message-Digest algorithm 5 is one of the most popular ways to get the message digest. Received shortcuts can be used as digital signatures of files or strings. This paper contains the implementation of this algorithm in C++ programming language. You can find there the description of the class, which can serve as a library in different programs written in C++.*

Keywords: *Cryptography, Message-Digest, digital signature, C++*

1. WSTĘP

Zadaniem funkcji MD5 jest generacja z ciągu liczb o dowolnej długości, 128-bitowego skrótu. Takie skróty mogą być wykorzystane do podpisu cyfrowego. Podpis cyfrowy stosuje się w kryptografii do identyfikacji, weryfikacji oraz zabezpieczenia wiadomości elektronicznych, dokumentów i plików przed nieuprawnioną modyfikacją i fałszerstwem.

2. FUNKCJA SKRÓTU MD5

Algorytm funkcji MD5 został opracowany przez Rona Rivesta w 1991 roku. Algorytm MD5 przetwarza dane wejściowe w 512-bitowych blokach, podzielonych na szesnaście 32-bitowych podbloków. Po przetworzeniu algorytmu otrzymywany jest zbiór czterech bloków po 32 bity każdy. Połączone ze sobą bloki utworzą jeden ciąg o długości 128 bitów. Ten ciąg jest skrótem [2].
W 2004 roku powstał projekt MD5CRK, którego twórcą był Jean-Luc Cooke i jego współpracownicy. Celem projektu było wykazanie możliwości kolizji, czyli wyznaczenie wiadomości o takiej samej wartości skrótu co zadana. Wykorzystując globalną sieć oraz dużą liczbę komputerów udało się wykazać, że podrabianie podpisów cyfrowych jest możliwe.

17 sierpnia 2004 chińscy naukowcy Xiaoyun Wang, Dengguo Fen, Xuejia Lai i Hongbo Yu opublikowali analityczny algorytm ataku na funkcję MD5. Do podrobienia podpisu cyfrowego wystarczała godzina działania komputera IBM P690, w pełni ukazało to słabość algorytmu[1].

Niespełna rok później, w marcu 2005 czeski kryptolog Vlastimil Klíma opublikował algorytm, który potrafił znaleźć kolizję podpisu cyfrowego w ciągu minuty. W tym celu użył metody tunnelingu.

Opis algorytmu:

1. Uzupełnienie wiadomości wejściowej – polega na dodaniu bitu o wartości 1 a następnie tylu zer, aby ciąg składał się z 512 bitowych bloków i ostatniego, mniejszego od pozostałych o 64 bity, 448 bitowego bloku.

2. Do otrzymanego zestawu bloków dodajemy 64 bitowy blok reprezentujący rozmiar wiadomości. Dzięki temu zabiegowi otrzymany został ciąg znaków będący wielokrotnością 512 bitowych bloków. Różne wiadomości po dodaniu ciągu uzupełniającego nie będą miały tej samej postaci.
3. Zdefiniowane zostają cztery zmienne: A, B, C i D, każda o długości 32 bitów. Przypisane są do nich następujące wartości zapisane w systemie heksadecymalnym:

   A = 01 23 45 67
   B = 89 ab cd ef
   C = fe dc ba 98
   D = 76 54 32 10

4. W tym kroku rozpoczyna się główna pętla algorytmu. Jest ona realizowana dla wszystkich 512-bitowych bloków, które zawiera wiadomość. Zmienne są kopiowane na cztery inne zmienne: A na AA, B na BB, C na CC i D na DD. Główna pętla algorytmu składa się z czterech podobnych cykli, w których wykonywane jest po 16 operacji. W każda operacja oblicza nieliniową funkcję trzech z czterech zmiennych: A, B, C i D. Do wyniku zostaje dodana wartość pozostałej zmiennej, podblok i stała. Otrzymany wynik przesuwany jest cyklicznie w prawo o zmienną liczbę bitów i sumowany z jedną ze zmiennych.
   Algorytm korzysta z następujących funkcji nieliniowych, przypisanych do właściwych cykli:
   F(X, Y, Z) = XY or (not X)Z
   G(X, Y, Z) = XZ or Y(not Z)
   H(X, Y, Z) = X xor Y xor Z
   I(X, Y, Z) = Y xor (X or (not Z))

   Ponadto stosuje się 64-elementową tablicę T wyznaczaną ze wzoru:

   $$T_i = 2^{32} \cdot |\sin(i)| = 4294967296 \cdot |\sin(i)|$$

   Jeśli M_i reprezentuje podblok j wiadomości i $<<<s$ oznacza przesunięcie w lewo o s bitów, to przebieg algorytmu dla poszczególnych etapów wygląda następująco:

   ETAP I
   FF(a, b, c, d, M_j, s, T_i)
   $a = b + ((a + F(b, c, d) + M_j + T_i) <<< s)$
   ETAP II
   GG(a, b, c, d, M_j, s, T_i)
   $a = b + ((a + G(b, c, d) + M_j + T_i) <<< s)$
   ETAP III
   HH(a, b, c, d, M_j, s, T_i)
   $a = b + ((a + H(b, c, d) + M_j + T_i) <<< s)$
   ETAP IV
   II(a, b, c, d, M_j, s, T_i)
   $a = b + ((a + I(b, c, d) + M_j + T_i) <<< s)$

Po wykonaniu powyższych operacji do wartości zmiennych A, B, C i D są dodawane analogicznie wartości zmiennych AA, BB, CC i DD, następnie algorytm przetwarza kolejny blok wiadomości. Wynikiem algorytmu jest połączenie 32-bitowych wartości zmiennych A, B, C oraz D[3].

3. IMPLEMENTACJA ALGORYTMU W JĘZYKU C++

W celu zaimplementowania algorytmu funkcji MD5, zaprojektowano wzorzec klasy [4]:

```
class md5
{
public:
   md5() { Init(); }
   void  Init();
   void  Update(uchar* chInput, uint4 nInputLen);
   void  Finalize();
   uchar*   Digest() { return m_Digest; }

private:

   void  Transform(uchar* block);
   void  Encode(uchar* dest, uint4* src, uint4
nLength);
   void  Decode(uint4* dest, uchar* src, uint4
nLength);


   inline  uint4  rotate_left(uint4 x, uint4 n)
                    { return ((x << n) | (x >>
(32-n))); }

   Inline  uint4  F(uint4 x, uint4 y, uint4 z)
                    { return ((x & y) | (~x & z));
}

   inline  uint4  G(uint4 x, uint4 y, uint4 z)
                    { return ((x & z) | (y & ~z));
}

   inline  uint4  H(uint4 x, uint4 y, uint4 z)
                    { return (x ^ y ^ z); }

   inline  uint4  I(uint4 x, uint4 y, uint4 z)
                    { return (y ^ (x | ~z)); }
```

```
   inline   void  FF(uint4& a, uint4 b, uint4 c,
uint4 d, uint4 x, uint4 s, uint4 ac)
                    { a += F(b, c, d) + x + ac; a
= rotate_left(a, s); a += b; }

   inline   void  GG(uint4& a, uint4 b, uint4 c,
uint4 d, uint4 x, uint4 s, uint4 ac)
                    { a += G(b, c, d) + x + ac; a
= rotate_left(a, s); a += b; }

   inline   void  HH(uint4& a, uint4 b, uint4 c,
uint4 d, uint4 x, uint4 s, uint4 ac)
                    { a += H(b, c, d) + x + ac; a
= rotate_left(a, s); a += b; }

   inline   void  II(uint4& a, uint4 b, uint4 c,
uint4 d, uint4 x, uint4 s, uint4 ac)
                    { a += I(b, c, d) + x + ac; a
= rotate_left(a, s); a += b; }
```

Klasa md5 zawiera w sobie cały przebieg algorytmu w formie kodu. Do działania kodu niezbędne są poniższe funkcje deklarujące jego przebieg.

Aby zainicjować nowy kontekst niezbędne jest jego wywołanie przy użyciu poniższej funkcji:

```
void md5::Init()
{
   memset(m_Count, 0, 2 * sizeof(uint4));

   m_State[0] = 0x67452301;
   m_State[1] = 0xefcdab89;
   m_State[2] = 0x98badcfe;
   m_State[3] = 0x10325476;
}
```

Następnie dokonywana jest operacja aktualizacji wprowadzonej wiadomości poprzez dodanie 64-bitowego bloku.

```
void md5::Update(uchar* chInput, uint4 nInputLen)
{
   uint4 i, index, partLen;

   index = (unsigned int)((m_Count[0] >> 3) &
0x3F);

   if ((m_Count[0] += (nInputLen << 3)) <
(nInputLen << 3))
      m_Count[1]++;

   m_Count[1] += (nInputLen >> 29);

   partLen = 64 - index;

   if (nInputLen >= partLen)
   {
      memcpy( &m_Buffer[index], chInput, partLen
);
      Transform(m_Buffer);

      for (i = partLen; i + 63 < nInputLen; i +=
64)
         Transform(&chInput[i]);

      index = 0;
   }
   else
      i = 0;

  memcpy( &m_Buffer[index], &chInput[i],
nInputLen-i );
}
```

Po zaktualizowaniu wiadomości dokonywana jest podstawowa transformacja przy użyciu tablicy.

```
void md5::Transform (uchar* block)
{
  uint4 a = m_State[0], b = m_State[1], c =
m_State[2], d = m_State[3], x[16];

  Decode (x, block, 64);

  // Runda 1
  FF (a, b, c, d, x[ 0], S11, 0xd76aa478);
  FF (d, a, b, c, x[ 1], S12, 0xe8c7b756);
  FF (c, d, a, b, x[ 2], S13, 0x242070db);
  FF (b, c, d, a, x[ 3], S14, 0xc1bdceee);
  FF (a, b, c, d, x[ 4], S11, 0xf57c0faf);
  FF (d, a, b, c, x[ 5], S12, 0x4787c62a);
  FF (c, d, a, b, x[ 6], S13, 0xa8304613);
  FF (b, c, d, a, x[ 7], S14, 0xfd469501);
  FF (a, b, c, d, x[ 8], S11, 0x698098d8);
  FF (d, a, b, c, x[ 9], S12, 0x8b44f7af);
  FF (c, d, a, b, x[10], S13, 0xffff5bb1);
  FF (b, c, d, a, x[11], S14, 0x895cd7be);
  FF (a, b, c, d, x[12], S11, 0x6b901122);
  FF (d, a, b, c, x[13], S12, 0xfd987193);
  FF (c, d, a, b, x[14], S13, 0xa679438e);
  FF (b, c, d, a, x[15], S14, 0x49b40821);

 // Runda 2
  GG (a, b, c, d, x[ 1], S21, 0xf61e2562);
  GG (d, a, b, c, x[ 6], S22, 0xc040b340);
  GG (c, d, a, b, x[11], S23, 0x265e5a51);
  GG (b, c, d, a, x[ 0], S24, 0xe9b6c7aa);
  GG (a, b, c, d, x[ 5], S21, 0xd62f105d);
  GG (d, a, b, c, x[10], S22,  0x2441453);
  GG (c, d, a, b, x[15], S23, 0xd8a1e681);
  GG (b, c, d, a, x[ 4], S24, 0xe7d3fbc8);
  GG (a, b, c, d, x[ 9], S21, 0x21e1cde6);
  GG (d, a, b, c, x[14], S22, 0xc33707d6);
  GG (c, d, a, b, x[ 3], S23, 0xf4d50d87);
  GG (b, c, d, a, x[ 8], S24, 0x455a14ed);
  GG (a, b, c, d, x[13], S21, 0xa9e3e905);
  GG (d, a, b, c, x[ 2], S22, 0xfcefa3f8);
  GG (c, d, a, b, x[ 7], S23, 0x676f02d9);
  GG (b, c, d, a, x[12], S24, 0x8d2a4c8a);

  // Runda 3
  HH (a, b, c, d, x[ 5], S31, 0xfffa3942);
```

```
  HH (d, a, b, c, x[ 8], S32, 0x8771f681);
  HH (c, d, a, b, x[11], S33, 0x6d9d6122);
  HH (b, c, d, a, x[14], S34, 0xfde5380c);
  HH (a, b, c, d, x[ 1], S31, 0xa4beea44);
  HH (d, a, b, c, x[ 4], S32, 0x4bdecfa9);
  HH (c, d, a, b, x[ 7], S33, 0xf6bb4b60);
  HH (b, c, d, a, x[10], S34, 0xbebfbc70);
  HH (a, b, c, d, x[13], S31, 0x289b7ec6);
  HH (d, a, b, c, x[ 0], S32, 0xeaa127fa);
  HH (c, d, a, b, x[ 3], S33, 0xd4ef3085);
  HH (b, c, d, a, x[ 6], S34,  0x4881d05);
  HH (a, b, c, d, x[ 9], S31, 0xd9d4d039);
  HH (d, a, b, c, x[12], S32, 0xe6db99e5);
  HH (c, d, a, b, x[15], S33, 0x1fa27cf8);
  HH (b, c, d, a, x[ 2], S34, 0xc4ac5665);

  // Runda 4
  II (a, b, c, d, x[ 0], S41, 0xf4292244);
  II (d, a, b, c, x[ 7], S42, 0x432aff97);
  II (c, d, a, b, x[14], S43, 0xab9423a7);
  II (b, c, d, a, x[ 5], S44, 0xfc93a039);
  II (a, b, c, d, x[12], S41, 0x655b59c3);
  II (d, a, b, c, x[ 3], S42, 0x8f0ccc92);
  II (c, d, a, b, x[10], S43, 0xffeff47d);
  II (b, c, d, a, x[ 1], S44, 0x85845dd1);
  II (a, b, c, d, x[ 8], S41, 0x6fa87e4f);
  II (d, a, b, c, x[15], S42, 0xfe2ce6e0);
  II (c, d, a, b, x[ 6], S43, 0xa3014314);
  II (b, c, d, a, x[13], S44, 0x4e0811a1);
  II (a, b, c, d, x[ 4], S41, 0xf7537e82);
  II (d, a, b, c, x[11], S42, 0xbd3af235);
  II (c, d, a, b, x[ 2], S43, 0x2ad7d2bb);
  II (b, c, d, a, x[ 9], S44, 0xeb86d391);

  m_State[0] += a;
  m_State[1] += b;
  m_State[2] += c;
  m_State[3] += d;

  memset(x, 0, sizeof(x));
}
```

Po dokonaniu powyższych transformacji algorytm zostaje zakończony. Finalizacja odbywa się poprzez zwrócenie skrótu wiadomości i wyzerowanie kontekstu.

```
void md5::Finalize()
{
   uchar bits[8];
   uint4 index, padLen;

   Encode (bits, m_Count, 8);

   index = (unsigned int)((m_Count[0] >> 3) &
0x3f);
   padLen = (index < 56) ? (56 - index) : (120 -
index);
   Update(PADDING, padLen);

   Update (bits, 8);

   Encode (m_Digest, m_State, 16);

   memset(m_Count, 0, 2 * sizeof(uint4));
   memset(m_State, 0, 4 * sizeof(uint4));
   memset(m_Buffer,0, 64 * sizeof(uchar));
}
```

Zakończony algorytm można wywołać poniższymi funkcjami.

```
char* PrintMD5(uchar md5Digest[16])
{
   char chBuffer[256];
   char chEach[10];
   int nCount;

   memset(chBuffer,0,256);
   memset(chEach, 0, 10);

   for (nCount = 0; nCount < 16; nCount++)
   {
      sprintf(chEach, "%02x", md5Digest[nCount]);
      strncat(chBuffer, chEach, sizeof(chEach));
   }
```

Przedstawiona powyżej funkcja PrintMD5 konwertuje wynik zakończonego algorytmu do zmiennej typu char*.

```
char* MD5String(char* szString)
{
   int nLen = strlen(szString);
   md5 alg;

   alg.Update((unsigned char*)szString, (unsigned
int)nLen);
   alg.Finalize();

   return PrintMD5(alg.Digest());

}
```

MD5String wykonuje algorytm na zmiennej char* zwracając wynik, którym jest gotowy kod, w postaci tej zmiennej. char* może być łatwo konwertowany do innego typu zmiennej.

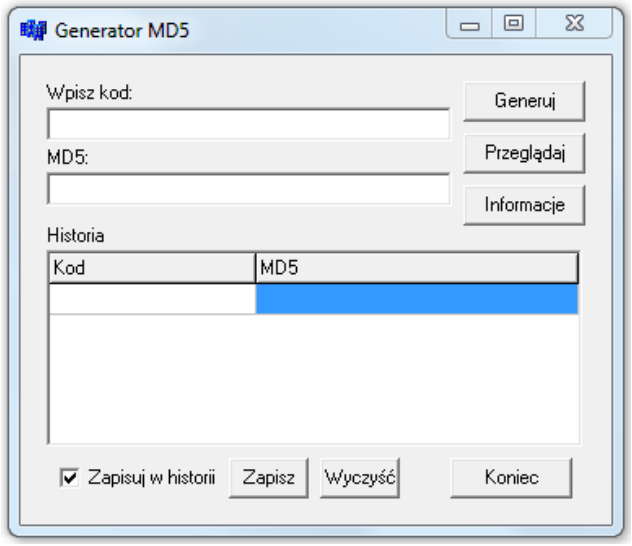
```
char* MD5File(char* szFilename)
{
   FILE* file;
   md5 alg;
   int nLen;
   unsigned char chBuffer[1024];

   try
   {
      memset(chBuffer, 0, 1024);

      if ((file = fopen (szFilename, "rb")) !=
NULL)
      {
```

```
        while (nLen = fread (chBuffer, 1, 1024,
file))
        alg.Update(chBuffer, nLen);
        alg.Finalize();
        fclose (file);
        return PrintMD5(alg.Digest());
      }
   }
   catch(...)
   {
   }
   return NULL;
}
```

MD5File realizuje algorytm dla plików. Jeżeli wprowadzone dane pozwolą na wykonanie algorytmu, wynik jest wyświetlany w formie char*. W przypadku, gdy podany plik jest nieprawidłowy funkcja nie zwraca wyniku.



Rysunek. 1 Interfejs przykładowej aplikacji wykorzystującej klasę `md5`.

Powyższy rysunek przedstawia interfejs przykładowej aplikacji wykorzystującej klasę md5. Jej celem jest zilustrowanie działania algorytmu funkcji skrótu MD5. Aplikacja ta korzysta z wyżej opisanych funkcji, pozwalających na wygenerowane hashu z tekstu oraz plików. Wynik algorytmu jest wyświetlany w polu MD5.

Taka aplikacja może służyć nie tylko do zademonstrowania działania algorytmu, ale może być także podstawą do tworzenia innych, bardziej skomplikowanych programów.

4. BADANIA

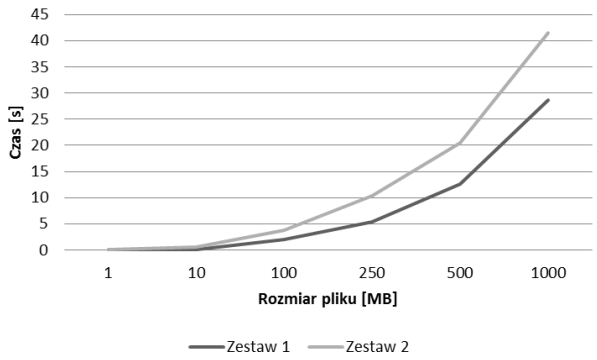
Badania polegają na wygenerowaniu skrótu wiadomości z plików testowych o rozmiarach 1MB, 10MB, 100MB, 250MB, 500MB i 1000MB oraz porównaniu czasu działania funkcji dla poszczególnych plików na dwóch różnych platformach testowych.

Pierwsza platforma testowa to komputer wyposażony w czterordzeniowy procesor Intel Core i7-2630QM o taktowaniu 2GHz na rdzeń oraz 4GB pamięci RAM, działający pod kontrolą 64-bitowego systemu operacyjnego Windows 7.

Druga platforma testowa to komputer wyposażony w dwurdzeniowy procesor Intel Celeron Dual-Core T3500 o taktowaniu 2,10GHz na rdzeń oraz 2GB pamięci RAM, działający pod kontrolą 32-bitowego systemu operacyjnego Windows 7.

Wyniki badań przedstawia poniższa tabela i wykres.

Rozmiar pliku [MB]	**Zestaw testowy 1 [s]**	**Zestaw testowy 2 [s]**
1	0,001	0,001
10	0,001	0,5
100	2	3,8
250	5,4	10,4
500	12,6	20,5
1000	28,7	41,5



Na powyższym wykresie można zaobserwować zależność, przedstawiającą czas w jakim każdy z zestawów wygenerował skrót wiadomości z pliku.

W pierwszej, mocniejszej platformie testowej skróty wiadomości plików o rozmiarach 1MB i 10MB zostały wygenerowane z tą samą prędkością. Przy większych plikach można zaobserwować rosnący czas działania algorytmu.

Druga, słabsza platforma gorzej radziła sobie z działaniem funkcji. Różnica jest widoczna już pomiędzy plikami o najmniejszych rozmiarach, gdzie 10MB plik został wygenerowany w zauważalnie dłuższym czasie. Przy

250MB pliku drugi zestaw generował skrót wiadomości prawie dwa razy dłużej.
Jak widać algorytm dla coraz większych plików wydłuża swoje działanie, jednak prędkość komputera na którym będzie generowany hash również ma istotne znaczenie.

5. PODSUMOWANIE I WNIOSKI

W niniejszym artykule przedstawiono algorytm funkcji skrótu MD5, a także jego implementację w języku C++. Nie zamieszczono kodu źródłowego niektórych funkcji, ale są one stosunkowo proste do odtworzenia, dlatego nie istnieje po prostu taka potrzeba. Dla zainteresowanych autor udostępni pełny kod źródłowy drogą mailową.
Pomimo słabości MD5 jest on nadal stosowany. Podpisy cyfrowe generowane z tego algorytmu są wykorzystywane przy sprawdzeniu autentyczności plików znajdujących się w Internecie.

Funkcja skrótu MD5 to stosunkowo prosty algorytm służący do generacji 128-bitowego skrótu. Jak widać jest on również łatwy do implementacji.

Literatura

1. Xiaoyun Wang, Dengguo Feng, Xuejia Lai, Hongbo Yu. „Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD" Cryptology ePrint Archive Report 2004/199
2. J. Pieprzyk, T. Hardjono, J. Seberry „Teoria bezpieczeństwa systemów komputerowych", Helion, Gliwice 2005
3. Schneier Bruce, „Applied Cryptography, Second Edition: Protocols, Algorthms, and Source Code in C", John Wiley & Sons, Inc. 1996
4.http://www.codeguru.com/cpp/cpp/algorithms/article.php/c5087/Implementing-the-MD5-Algorithm.html