Kryptografia – Krzywe eliptyczne

(Krzywe eliptyczne i ich zastosowanie w kryptografii)

na przykład w:

„An Introduction to the
Theory of Elliptic Curves”

Joseph H. Silverman
Brown University and
NTRU Cryptosystems, Inc.

Kryptologia – zagrożenia dla metody RSA

Funkcja dzeta

$$\zeta(x) = \sum_{n=1}^{\infty} \frac{1}{n^x}$$

Funkcja dzeta Riemanna (jej zera są związane z liczbami pierwszymi!)

$$\zeta(z) = \frac{1}{1-2^{1-z}} \sum_{n=0}^{\infty} \frac{1}{2^{n+1}} \sum_{k=0}^{\infty} (-1)^k \binom{n}{k} (k+1)^{-z}$$

Kryptologia – zagrożenia dla metody RSA

Ciekawostka:

Problemy Hilberta to lista 23 zagadnień matematycznych przedstawiona przez Davida Hilberta na Międzynarodowym Kongresie Matematyków w Paryżu w 1900 roku podczas referatu pokazującego stan matematyki na przełomie XIX i XX wieku.

Zagadnienie nr 8 dotyczy funkcja dzeta Riemanna (do dziś problem nie został rozwiązany)

Kryptologia – zagrożenia dla metody RSA

Każdą liczbę naturalną można przedstawić jako iloczyn pewnych liczb pierwszych i że rozkład ten jest jednoznaczny. Jest to tzw. „Podstawowe twierdzenie arytmetyki”, z którego wynika, że liczby pierwsze to swego rodzaju cegiełki służące do budowania innych liczb naturalnych.. Nie było sprawą oczywistą ile jest liczb pierwszych. Euklides jako pierwszy udowodnił, że w istocie jest ich nieskończenie wiele.

Kryptologia – zagrożenia dla metody RSA

Rozumowanie Euklidesa:

Załóżmy, że jest skończenie wiele liczb pierwszych, dajmy na to n. Oznaczmy je następująco: p1, p2, p3 ,... pn. Rozważmy teraz liczbę W= p1 *p2 *p3 *...* pn +1. Żadna z liczb p1, p2, p3 ,... pn nie jest dzielnikiem liczby W (bo jest dzielnikiem liczby W-1). Zatem muszą istnieć jeszcze inne liczby pierwsze będące dzielnikami liczby W (być może samo W jest pierwsze), co oczywiście przeczy temu, że liczb pierwszych jest skończenie wiele.

Wniosek: liczb pierwszych jest nieskończenie wiele (czyli ich nie zabraknie).

Podpis cyfrowy

Podpis cyfrowy jest odpowiednikiem tradycyjnego podpisu, składanego na dokumentach. Własności jego to:

- jedynie osoba X może odtworzyć podpis osoby X – podrobienie niewykonalne

- pod danym dokumentem – kopiowanie podpisu z jednego dokumentu na drugi powinno być NIEWYKONALNE

Podpis cyfrowy

Algorytm realizacji:

1. Osoba A jest w posiadaniu prywatnego klucza szyfrującego PKS. Jednocześnie pasujący do PKS publiczny klucz deszyfrujący PKD i algorytm użycia kluczy jest ogólnie znany.
2. Podpisanie listu L : Osoba A generuje kryptogram z oryginalnego listu za pomocą PKS. Kryptogram ten jest przekazywany do adresata jako podpis osoby A. Oryginalny list jest przesyłany łącznie z podpisem.
3. Osoba która chce się przekonać o prawdziwości podpisu deszyfruje kryptogram za pomocą PKD; następnie porównuje oryginalny list ze zdeszyfrowanym kryptogramem.

Podpis cyfrowy

Możliwa wada: przedstawiony sposób ma tą wadę, że podpis jest długi – co najmniej tak długi jak podpisywany dokument.

Gwarantowane jest jednak to, że podpis nie zostanie przeniesiony na inny dokument ! (no bo co fałszerz miałby przenosić?)

Jak generować krótkie podpisy...?

Podpis cyfrowy

Sposobem na generowanie krótkich podpisów jest podpisywanie nie oryginalnego list L, lecz wartości H(L) !

H jest jednokierunkową funkcją hashującą (”mieszającą”). Podaje jej się zbiór L jako argument, na tej podstawie produkuje ona wartość H(L).

Znane algorytmy: md2, md4, md5, sha1 .

Termin angielski na określenie H(L) : ”message digest”

np. man sha1 dostarczy krótkiego opisu

(UNIX posiada np. komendy sha1sum oraz md5sum)

Podpis cyfrowy

SHA1 wylicza skondensowaną reprezentację zbioru. Dla zbioru o rozmiarze nie wiekszym niż 2**64 bitów, algorytm ten produkuje 160-bitową wartość. Uważa się, że SHA1 jest bezpieczny, tzn. jest bardzo kosztowne obliczeniowo znalezienie innego zbioru, który dałby identyczną 160-bitową wartość.

Podpis cyfrowy

MD5 tworzy dla każdego pliku wejściowego 128-bitowy „odcisk palca” czy „message digest”

Podpis cyfrowy

Niektóre algorytmy podpisu cyfrowego:

- ECDSA (Elliptic Curve Digital Signature Algorithm)
- ESIGN
- FLASH
- QUARTZ
- RSA
- RSA-PSS
- SFLASH

infinity(nieskończoność) oraz „not-a-number”

GNU C używa standardu IEEE 754 dla reprezentacji liczb zmiennoprzecinkowych. Standard ten definiuje pięć sytuacji wyjątkowych (”exception”).

1.Nieprawidłowa operacja (dodawanie, odejmowanie nieskończoności, dzielenie zera przez zero, dzielenie nieskończoności przez nieskończoność, sqrt z –1, ogólnie każda funkcja matematyczna wyliczana od argumentu nie należącej do dziedziny tej funkcji. Wynikiem jest NAN.

2.Dzielenie przez zero (skończona niezerowa wartość jest dzielona przez zero). Wynikiem jest ”plus nieskończoność” albo ”minus nieskończoność”, w zależności od znaku operandów.

3.Nadmiar (”overflow”). Gdy wynik nie może być przedstawiony jako wartość skończona.

4. Niedomiar (”underflow”). Gdy rezultat jest zbyt mały by być prezentowany z wymaganą dokładnością; wynikiem jest nieprecyzyjna mała wartość lub zero

infinity(nieskończoność) oraz „not-a-number”

5.Niedokładność (”inexact”). Jest sygnalizowana wtedy, gdy wynik nie jest dokładny (np. wyliczanie pierwiastka z liczby 2.)

infinity(nieskończoność) oraz „not-a-number”

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
   int main ()
{  float a,b,c;
    a=1./0.;
    b=log(0.);
    c=sqrt(-1);
  printf("\n a=%f b=%f c=%f\n",a,b,c);
}    /*  wypisze:        a=inf   b=-inf   c=nan      program wykona się!*/
```

przykład: wartości ”inf” oraz ”nan”

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

  int main ()
{  int k;
  int *wsk1, *wsk2, *wsk3, *wsk4;
  char * w;
  float f;
  float f1,f2,f3,f4;
```

przykład: wartości ”inf” oraz ”nan”

```
printf(" wartosc  HUGE_VAL  i  -HUGE_VAL :%f
%f\n",HUGE_VAL, -HUGE_VAL);  /* wypisze  inf  i  -inf  */

  f=pow(10.,300) * pow(10.,300);

  printf("\n   znowu duza liczba f:%f\n\n",f);

  f1=HUGE_VAL;          f2=f1*0.0;

  wsk1=(int *) &f1;

  wsk2=(int *) &f2;

  wsk3=(int *) &f3;

  wsk4=(int *) &f4;

  printf("\n f1=%f  f2=%f\n",f1,f2); /*  wypisze inf  i  nan  */

  printf("\n  f1=%x  f2=%x\n\n", *wsk1,*wsk2);
```

przykład: wartości ”inf” oraz ”nan”

```
/* zbudowanie float o wartosci  inf  oraz o wartosci nan*/

   printf("\n liczby dodatnia nieskonczonosc i NAN budowane
\"recznie\"\n");

      f3=0;

      w = (char *) &f3;

      *(w+0) =  0x00;       *(w+1) =  0x00;

      *(w+2) =  0x80;       *(w+3) =  0x7f;

            w = (char *) &f4;

            *(w+0) =  0x00;          *(w+1) =  0x00;

            *(w+2) =  0xc0;          *(w+3) =  0xff;
```

przykład: wartości ”inf” oraz ”nan”

```
   printf("\n f3=%f  f4=%f\n",f3,f4);
   printf("\n  f3=%x  f4=%x\n\n", *wsk3,*wsk4);


   exit(0);
}/* koniec funkcji  main  */
```

poniżej wynik przedstawionego program (produkuje wartości ”inf” oraz ”nan”)

wartosc HUGE_VAL i -HUGE_VAL :inf -inf

znowu duza liczba f:inf

f1=inf f2=nan

f1=7f800000 f2=ffc00000

liczby dodatnia nieskonczonosc i NAN budowane "recznie"

f3=inf f4=nan

f3=7f800000 f4=ffc00000

standard IEEE dla liczb zmiennoprzecinkowych

S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF = V

1)jeśli E=255 i F różne od zera, to V = ”not-a-number”

2)jeśli E=255 i F=0 i S=1 to V = ”minus nieskończoność”

3)jeśli E=255 i F=0 i S=0 to V = ”plus nieskończoność”

4)jeśli 0<E<255 to V=(-1)**S * 2 ** (E-127) * 1.F

5)jeśli E=0 i F różne od zera, to V=(-1)**S * 2 ** (E-126) * 0.F

6)jeśli E=0 i F=0 i S=1, to V = -0

7)jeśli E=0 i F=0 i S=0, to V = 0

(są dwie reprezentacje zera !)

standard IEEE: dwa zmiennoprzecinkowe zera

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
    int main ()
{           float a=1.,b=-1.,c,d;
            int *wsk1, *wsk2, *wsk3, *wsk4;
   wsk1=(int*)&a; wsk2=(int*)&b;wsk3=(int*)&c;wsk4=(int*)&d;
   c=a/HUGE_VAL;     d=b/HUGE_VAL;
   printf("a,b,c,d = %f %f %f %f\n",a,b,c,d);
   printf("a,b,c,d = %x %x %x %x\n",*wsk1,*wsk2,*wsk3,*wsk4);
   exit(0);       }/* koniec funkcji  main  (wynik na
                       następnym  slajdzie)   */
```

standard IEEE: dwa zmiennoprzecinkowe zera

a,b,c,d = 1.000000 -1.000000 0.000000 -0.000000

a,b,c,d = 3f800000 bf800000 0 80000000

”szerokie znaki” czyli ”wide characters”

Jednym z problemów dotyczących wewnętrznej reprezentacji znaków jest ich ilość. Niestety, 8 bitów to za mało. W związku z tym także w różnych wersjach języka C, także w gcc, wprowadzone zostały dwa dodatkowe typy.

wchar_t (odpowiednik char)

wint_t (odpowiednik int)

w przypadku gcc oba te typy mają długość 4 bajtów.

typ ten oryginalnie jest zdefiniowany w **wchar.h**

(dołączane przez **#include <wchar.h>**)

(skoro 1 bajt to za mało, naturalny wybór to 2 bajty lub 4 bajty)

”szerokie znaki” czyli ”wide characters”

Najczęściej używanym zbiorem znaków dla takich międzynarodowych szerokich znaków jest UNICODE oraz ISO 10646 (zwany także Universal Character Set).

Istnieje UCS-2 (16-bitowy) oraz UCS-4 (32-bitowy). UCS-4 może reprezentować każdy znak UNICODE.

Jest jeszcze UTF-8 gdzie znaki ASCII są reprezentowane przez pojedyncze bajty ASCII, zaś znaki nie-ASCII przez sekwencje 2-6 bajtów.

Są w gcc funkcje które operują na ”wide characters”, kopiujące te znaki do ich reprezentacji wielobajtowej, i odwrotnie. Tylko – działanie tych funkcji zależy od wybranego kodowania znaków w systemie operacyjnym.

”szerokie znaki” czyli ”wide characters”

Informacje o Unicode są łatwo dostępne, np. po polsku na stronie www.unikod.pl

”szerokie znaki” czyli ”wide characters”

np. funkcje mbrtowc( ) multibyte → ”wide characters”

wcrtomb( ) ”wide characters” → multibyte

(np. mbrtowc to „**m**ulti**b**yte **r**estartable **to** **w**ide **c**haracter”)

dla jednobajtowych: btowc() byte → ”wide character”

wctob() ”wide character” → byte

są potrzebne np. po to, by móc czytać strumień bajtów i ładować kolejne ”wide characters” do kolejnych elementów macierzy typu wchar_t[ ]

”szerokie znaki” czyli ”wide characters”

size_t mbrtowc(wchar_t *pwc, const char *s, size_t n, mbstate_t *ps); /* to prototyp funkcji */

Opis można uzyskać np. z „man mbrtowc”.

Najważniejsza część opisu:
…mbrtowc function inspects at most n bytes of the multibyte string starting at s, extracts the next complete multi-byte character, converts it to a wide character and stores it at *pwc; if the converted wide character is not L’\0’, it returns the number of bytes that were consumed from s …

size_t mbrtowc(wchar_t *pwc, const char *s, size_t n, mbstate_t *ps); /* to prototyp funkcji */

…if error condition happens, mbrtowc function returns **(size_t) -2** or **(size_t) -1**…

Na zbiorze widech2.c przykład użycia mbrtowc().

Przykład pobrany z opisu mbrtowc() na stronie

http://www.gnu.org/software/libc/manual/html_node/index.html (tam jest hyperlink *Function Index: Index of functions and function-like macros*)

”klasyfikacja” zmiennych zmiennoprzecinkowych

Floating-Point Number Classification Functions

ISO C99 – definicja makra które umożliwia określenie, jakiego rodzaju jest dana wartość zmienno przecinkowa

int **fpclassify** (*float-type x*)

możliwe wartości zwracane:

FP_NAN

x jest ”not-a-number”

FP_INFINITE

x jest plus lub minus nieskończonością

FP_ZERO

x jest równe zero

”klasyfikacja” zmiennych zmiennoprzecinkowych

Floating-Point Number Classification Functions

int **fpclassify** (*float-type x*)

możliwe wartości zwracane:

FP_SUBNORMAL

liczby które są zbyt małe by reprezentować w formacie normalnym są prezentowane w formacie *”denormalized”*; format ten jest mnie dokładny ale może reprezentować wartości bliższe zeru

FP_NORMAL

dla wszystkich innych wartości x

(po jednym zawołaniu fpclassify() można rozpoznać sytuację)

”klasyfikacja” zmiennych zmiennoprzecinkowych

Floating-Point Number Classification Functions

ISO C99 – inne makra które umożliwiają określenie, jakiego rodzaju jest dana wartość zmiennoprzecinkowa (gcc –std=c99)

int **isfinite** (*float-type x*)

int **isnormal** (*float-type x*)

int **isnan** (*float-type x*)

int **isinf** (*float-type x*)

np. **isfinite**() jest równoważne:

```
(fpclassify (x) != FP_NAN && fpclassify (x) != FP_INFINITE)
```

przykład: użycie funkcji stat() i utime()

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <sys/stat.h>
#include <unistd.h>

    #include <sys/types.h>  /*  utime  */
    #include <utime.h>
    struct utimbuf alfa;
```

przykład: użycie funkcji stat() i utime()

```
int main()
{
  int n, ktrl;
  struct stat buf;
  n=time(NULL);
  printf("  n=%d\n",n);
  printf("czas odczytany:%s\n",ctime( (time_t *) &n));

ktrl=stat("test1", &buf);
if(ktrl==-1) {printf("\n nie ma zbioru...\n");exit(0);}
```

przykład: użycie funkcji stat() i utime()

```
printf("\n czas ostatniego dostepu     %d %s",
  buf.st_atime, ctime((time_t *) &buf.st_atime));
printf("\n czas ostatniej modyfikacji  %d %s",
  buf.st_mtime, ctime((time_t *) &buf.st_mtime));

alfa.actime = buf.st_atime + 10;
alfa.modtime= buf.st_mtime + 20;
 utime("test1", &alfa );   /*   modyfikacja czasów dostępu i modyfikacji */

printf("\n\n");
```

przykład: użycie funkcji stat() i utime()

```
stat("test1", &buf);
printf("\n czas ostatniego dostepu     %d %s",
  buf.st_atime, ctime((time_t *) &buf.st_atime));
printf("\n czas ostatniej modyfikacji  %d %s",
  buf.st_mtime, ctime((time_t *) &buf.st_mtime));



  exit(0);
}/*  koniec funkcji  main */
```

przykład: użycie funkcji stat() i utime()

n=1516053165
czas odczytany:Mon Jan 15 22:52:45 2018

czas ostatniego dostepu 1483994177 Mon Jan 9 21:36:17 2017

czas ostatniej modyfikacji 1483994207 Mon Jan 9 21:36:47 2017

czas ostatniego dostepu 1483994187 Mon Jan 9 21:36:27 2017

czas ostatniej modyfikacji 1483994227 Mon Jan 9 21:37:07 2017