

Bezpečnost a kryptografie

Projekt 1

Obsah

Příklad 1 - Vigeněrova šifra.....	1
Stavební kameny.....	1
Třídy v PHP.....	1
Referenční histogram anglického textu	1
Algoritmus prolomení	2
Výpočet podobnosti dvou histogramů.....	2
Nalezení délky klíče	2
Nalezení vlastního klíče když známe jeho délku.....	2
Odstranění opakování v klíči	3
Demonstrace	3
Šifrování komprimovaných dat	3
Příklad 2 - mikroDES	4
Implementace	4
Třídy	4
Ověření činnosti.....	4
Dešifrování textu	5
Znalosti	5
Řešení.....	5
Síla nebo inteligence?	6
Závěry	6
Příklad 3 - RSA 32	7
Třída	7
Generování klíčů.....	7
Algoritmus.....	7
Generování prvočísel	8
Výpočet inverzního prvku.....	9
Šifrování a dešifrování	10
Efektivita nad typem __int64.....	10
Literatura.....	11

Příklad 1 - Vigenerova šifra

Stavební kameny

Třídy v PHP

Pro Vigenerovu šifru jsem sestrojil šifrovací/dešifrovací třídu. Konstruktorem se zadává heslo a objekt pak poskytuje možnost práce s jednotlivými znaky nebo celými řetězci. Algoritmus je implementován dle zadání i když přednášky a informace z Internetu ukazují jenom použití na abecedě složené ze znaků A až Z.

```
class Vignener {
    function __construct( $key )
    public function reset()
    private function getKey()
    public function encryptChar( $inp )
    public function decryptChar( $inp )
    public function encrypt( $input )
    public function decrypt( $input )
}
```

Pro prolomení šifry se používá frekvenční analýza, takže další třída slouží na tvorbu a práci s histogramy. Histogram se vytvoří z dat zadaných v konstruktoru, taky se může upřesnit posunutí nebo perioda vzorkování vstupu. Po spočítání počtu znaků se graf normalizuje (hodnoty jsou pak z rozsahu 0 až 1). Čtení vstupu je přizpůsobeno výstupu Vigenerové šifry - např. znak za písmenem Z se počítá jako znak A, atd. Vyšší funkce histogramu pak slouží na výpočet podobnosti dvou histogramů - s volitelným vzájemným posunutím, zjištění podobnosti pro všechny možné posuvy naráz nebo zjištění průměrné podobnosti.

```
class histogram {
    public $freq = array();
    function __construct( $data = '', $offset = 0, $period = 1 )
    private function reset()
    private function normalize()
    public function show( $w = 75 )
    public function save()
    public function getSimilarity( histogram $x, $offset = 0 )
    public function getSimilarities( histogram $x )
    public function avgSimilarity( histogram $x )
}
```

Referenční histogram anglického textu

K úspěšnému prolomení šifry potřebujeme referenční histogram - po chvílce přemýšlení kde ho sehnat mě napadlo vyrobit si ho z nějakých textů. Protože knížky nejsou většinou volně přístupné a nechtěl jsem použít počítačovou literaturu, tak mě napadlo vyhledat scénáře k filmům a vygenerovat histogram z nich. Skript *make-reference-histogram.sh* provede vše potřebné:

```
#!/bin/sh
# Get script list (as HTML)
wget http://www.simplyscripts.com/full_movie.html
# Get target URL's (HTML processing parsing)
grep full_movie.html -e \\.txt \
| sed 's/[^\"]*href="//' \
| sed 's/".*//' \
| grep -v nbsp \
> script-list.txt
# Download the scripts ( about 71 MB )
mkdir data
cd data
wget -i ../script-list.txt
# Remove duplicates ( 71 MB -> 67 MB )
rm *.1
# Normalization ( 67 MB -> 48 MB )
cd ..
cat data/* | ./normalize.php >plaintext-sample-en.txt
# Make histogram for use in PHP
./php-histogram.php <plaintext-sample-en.txt >english-histogram.php
```

Algoritmus prolomení

Prolomení Vigeněrové šifry se děje ve dvou krocích:

- nalezení délky klíče
- nalezení klíče (s možností odstranění opakování)

Výpočet podobnosti dvou histogramů

Výpočet podobnosti je vlastně operace známá jako *konvoluce* a její implementace v mojí třídě umožňuje zadat ještě posunutí, které se použije třeba při detekci klíče v Césarově šifře:

```
public function getSimilarity( histogram $x, $offset = 0 ) {
    $sum = 0;
    $a = ord('A');
    for( $i=0; $i<=25; $i++ )
        $sum += $this->freq[$a+$i] * $x->freq[$a+($i+26-$offset)%26];
    return $sum;
}
```

Nalezení délky klíče

Pro nalezení délky musíme spočítat hodně histogramů - konkrétně provádíme cyklus od 1 do maximální hodnoty délky klíče (nastaveno na 32). V každém kroku tohoto cyklu známe tedy odhad délky (uložen v *\$L*) a provádíme výpočet průměrné hodnoty maximálních podobností histogramů pro všechny znaky klíče (posunutí v *\$O* jako 0 až *L-1*).

```
// maximal detection length
$max = 32;

// detect similarities
$sim = array();
for( $L = 1; $L <= $max; $L++ ) {
    $sum = 0;
    for( $O = 0; $O < $L; $O++ ) {
        $hist = new histogram( $cipher, $O, $L );
        $sum += $hist->avgSimilarity($english);
    }
    $sim[$L] = $sum/$L; // average similarity
}
```

Délka klíče je index minimální hodnoty v poli *\$sim*. Pokud použijeme první minimum, tak nemusíme nalézt správnou hodnotu, takže raději použijeme absolutní minimum pole. Pak sice nalezneme klíč který se opakuje, ale opakování odstraníme velice jednoduše, na rozdíl od použití špatné hodnoty délky klíče.

Nalezení vlastního klíče když známe jeho délku

Operace nalezení klíče je ta jednodušší při lámání Vigeněrové šifry, protože jde v podstatě jen o prolomení několik Césarových šifer. Pokud je známa délka klíče (*N*), tak pro zjištění vlastního klíče postupujeme následovně:

- vygenerujeme *N* histogramů ze vstupu takovým způsobem, že použijeme jenom každý *N*-tý znak při posunutí je 0 až *N-1*
- každý z těchto histogramů reprezentuje jedno písmeno šifry, takže porovnáme daný histogram s referenčním a zjistíme posunutí
- posunutí nám dává písmeno klíče

Detaily implementace - *\$L* je zadaná délka klíče (*N*). *\$cipher* je kousek začátku vstupu (aktuálně 8KB) což pro nalezení písmene je dostačující - aspoň se histogram počítá rychleji. *\$english* pak reprezentuje referenční histogram anglického textu. Vlastní kód pro nalezení klíče je:

```
for( $O = 0; $O < $L ; $O++ ) {
    $hist = new histogram($cipher,$O,$L); // histogram
    $sims = $hist->getSimilarities( $english ); // similarities
    $maxv = max($sims); // maximal value
    $offs = array_search($maxv,$sims); // index of that value
    echo chr( ord('A') + $offs - 1 ); // the character
}
```

Odstranění opakování v klíči

Pokud zadáme délku klíče několikanásobek skutečné délky, zjištěný klíč bude opakováním kratšího klíče. Tohle opakování lze odstranit pomocí jednoduchého algoritmu:

```
function unrepeat( &$text, $len ) {
    $l = strlen($text); // text length
    if (($l%$len)!=0) return false; // absolutely does not fit
    $num = $l / $len; // number of repeats
    $part = substr($text,0,$len); // part which is repeated
    // can be made up the text from repeating the part?
    if ($scan = $text===str_repeat($part,$num)) $text = $part;
    return $scan;
}

// unrepeat the password
for( $R = 1; $R <= $L; $R++ ) if (unrepeat($pass,$R)) break;
```

Demonstrace

Jako nezašifrovaný text si vezmeme scénář k filmu *Hackers*:

```
$ cp ./data/hackers.txt source.txt
```

Text se musí normalizovat (převést na velická písmena) a taky odstraníme mezery:

```
$ ./normalize.php <source.txt | ./removespaces.php >message.txt
```

Zakódujeme text zvoleným heslem:

```
$ ./encode.php EXAMPLEPASS <message.txt >ciphertext.txt
```

Zkusíme uhodnout délku klíče (hledá první minimum):

```
$ ./get-key-length.php <ciphertext.txt
1
```

To určitě nebude ta správná hodnota, takže provedeme hledání absolutního minima:

```
$ ./get-key-length.php --absolute <ciphertext.txt
22
```

Necháme uhodnout heslo:

```
$ ./guess-password.php 22 <ciphertext.txt
EXAMPLEPASS
```

Ve skutečnosti mělo heslo jenom 11 znaků, takže opakování bylo odstraněno.

Šifrování komprimovaných dat

Podle mě je použití substituční šifry na šifrování komprimovaných dat použitelné pokud jsou splněny tyto dvě podmínky:

- komprimování je vysoko účinné, takže histogram se přibližuje rovné čáře - tady je substituční šifra neodhalitelná, protože rotací nebo výměnou sloupců dostáváme znova tu samou rovnou čáru
- šifrují se pouze komprimovaná data bez hlaviček - ty totiž obsahují pořadí dat stejného charakteru (názvy souborů, zarovnávací kódy) které by mohly ulehčit prolomení

V praxi na tohle mysleli nejspíš i autoři programů - protože adresářovou strukturu lze rozbalit a prohlížet i bez znalosti klíče a po zadání nesprávného klíče se rozbalí pouze porušené soubory (případ ZIP-u). To že heslo bylo zadáno špatně zjistíme až z nesesdího CRC součtu.

Příklad 2 - mikroDES

Implementace

Třídy

Šifru mikroDES jsem implementoval v jazyce PHP5 pomocí dvojice tříd. První z nich se stará o vlastní algoritmus DES - zejména generování sub-klíčů K1, K2 a K3 a provedení šifrování nebo dešifrování pomocí těchto klíčů. Výpočet klíčů se děje v konstruktoru, vlastní šifrování je pak prováděno v režimu ECB (Electronic Code Book, každý blok nezávisle - stejným klíčem) a vždy se zpracovává jeden blok 6-ti bitů. Klíč je 9-ti bitový, sub-klíče jsou přímo trojice bitů.

```
class DES {
    function __construct( $key )
    function encrypt( $data )
    function decrypt( $data )
}
```

Uvedená třída DES využívá pro svoji činnost třídu která implementuje Feistelovu šifru. Tato třída provádí vždy jeden krok šifrování nebo dešifrování. Substituce pomocí SBOX-u je reprezentována polem, ve kterém je index vstupem SBox-u, hodnota v dané buňce je pak výstupem.

```
class Feistel {
    public $sbox = array(3,7,0,4,5,1,6,2);
    public $data;
    function __construct( $data )
    function forward( $key )
    function reverse( $key )
}
```

Dešifrování se provádí reverzním postupem, jak uvnitř Feistelové šifry, tak u třídy DES, která teda aplikuje klíče v opačném pořadí. Pro ověření činnosti jsem napsal ještě konverzi ASCII znaku na 6-ti bitové hodnoty (a zpět) a taky funkce na konverzi celého řetěze.

```
function s2i( $string )
function i2s( $ints )
function desEncrypt( $m, $key )
function desDecrypt( $c, $key )
```

Ověření činnosti

Ukázka že šifrování opravdu funguje je zde - zdrojový kód:

```
$key      = 0123;

$message = 'Text který bude zakodovan algoritmem microDES 9bit.';
$cipher  = desEncrypt( $message, $key );
$decoded = desDecrypt( $cipher, $key );

echo "message: $message\n";
echo "cipher:  $cipher\n";
echo "decoded: $decoded\n";
```

Výstup programu:

```
$ ./project.php
message: Text který bude zakodovan algoritmem microDES 9bit.
cipher:  XIDiOMiIJyOWTtIOMCMGtG CzOCS.GJdinInOndjJGExFOZwdi8
decoded: Text který bude zakodovan algoritmem microDES 9bit.
```

Protože šifra pracuje v režimu ECB, nemodifikuje frekvenční charakteristiku a je možný útok pomocí odhadu několika písmen a zbytek dohledat podle pořadí písmen ve slově (potřeba slovníku).

Dešifrování textu

Znalosti

Zašifrovaný text:

cvx5UXk1

Několik kombinací plaintext - ciphertext:

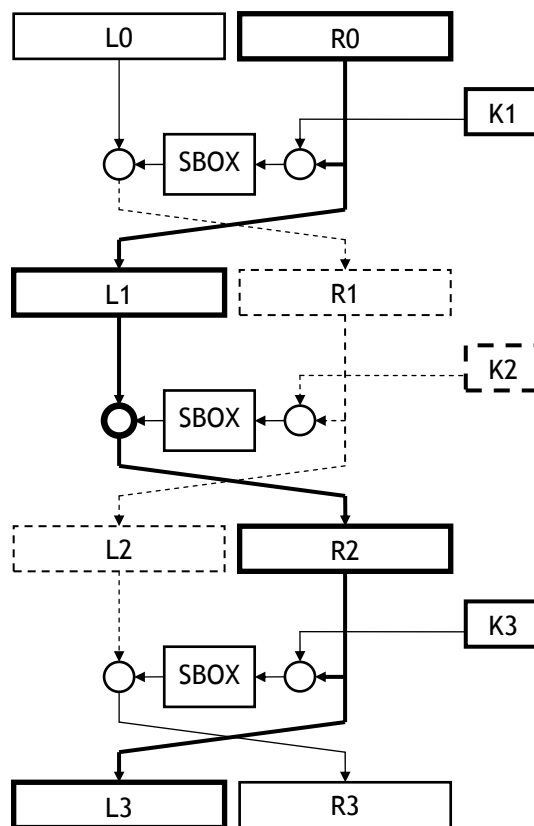
e	-> 0	100101	-> 011011
t	-> 5	110100	-> 111011
a	-> Q	100001	-> 010001
o	-> U	101111	-> 010101
space	-> 2	000000	-> 011101

Režim činnosti: ECB (Electronic Code Book) tj. každý znak je šifrován nezávisle na ostatních

Řešení

Šifru *mikroDES* jsem prolomil následovným způsobem. Jako první věc co mě napadla, byla hledat slabosti ve způsobu šifrování pomocí Feistelovy šifry, takže jsem si nakreslil všechny tři kola šifrování pod sebe (obr. 1) a začal dané schéma analyzovat.

První zjištění bylo, že když jsem dosadil jednu ze známých kombinací plaintext-ciphertext, mezeru a číslici 2 (L0R0 = 0000000, L3R3 = 011101) tak můžu spočítat výstup SBOX-u v druhém kole, protože znám 2 hodnoty ze 3 vystupujících v XOR-u, S2-out je teda 011. Vstupem pro SBOX tedy musí být S2-in = 000. Protože je před SBOX-em další XOR, tak musí platit že jeho vstupy jsou stejné - a tady to znamená, že $K2 == R1 == L2$ (čárkovaná čára). Po dosazení zbylých kombinací jsem zjistil, že je to stejně ve 3 případech z 5-ti.



Obr. 1: Prolomení šifry „mikroDES“

Další zjištění bylo, že pro kombinaci mezera a 2 je $R0 = 000$, takže vstupem prvního SBOX-u je přímo klíč K1. Levá část, je ale taky nulová ($L0 = 000$), takže čárkovaná čára, K2, bude muset nabýt hodnoty SBOX[K1].

V tomto místě už vím, že klíč není složen z náhodných hodnot, ale že je algoritmicky odvozen. Takže další bádání mělo za výsledek zjištění, že když L2 obsahuje rovnou klíč K2, lze přes XOR, reverzní SBOX a další XOR odvodit, že K3 je generován taky pomocí algoritmu.

Zde moje matematické schopnosti končili, protože praktické příklady výpočtů nad Galoissovým tělesem jsem nenašel ani na Internetu, takže jsem se rozhodl pro výpomoc pomocí algoritmů umělé inteligence.

Síla nebo inteligence?

Schéma zakreslené na obr.1 si lze představit jako elektrický obvod. Obsahuje dráty - (signály, proměnné) a obvody (funkce). Pokud provedeme analýzu tohoto druhu, zjistíme, že náš obvod DES obsahuje pouze tři typy funkcí:

- přímé propojení
- XOR
- SBOX

Přímé propojení a XOR jsou reverzibilní funkce - takže je jedno který konec (resp. konce) známe, ten neznámý lze dopočítat. U SBOX-u máme konkrétně v tomto případě štěstí, protože provádí mapování 8 prvků na 8 jiných, takže lze spočítat reverzní SBOX.

Postup řešení je následovní:

- definujeme obvod - signály a funkce napojené na mezi ně
- nastavíme signály do neurčitého stavu
- nastavíme známé signály (vstupy, výstupy)
- teď přijde na řadu „síla“, protože víme, že K2 a K3 bude vypočtené, ale neznáme K1, takže si uděláme odhad - budeme zkoušet dosazovat za K1 postupně 000 až 111
- tady nastupuje „inteligence“ - vyčíslení obvodu (provádí se v cyklu testování všech součástek tak dlouho, dokad' se nastavují neznámé hodnoty na známé)
- z obvodu přečteme hodnoty K2 a K3
- trojice K1K2K3 reprezentuje kandidátní klíč, poznačíme si ho, a zkusíme další K1

Z tohoto postupu dostaneme 8 kandidátních klíčů, takže jestli jsou platné - to nevíme ale můžeme to ověřit pomocí pětice známých dvojic. Provedeme šifrování známého písmene kandidátním klíčem, a pokud vytvořený zašifrovaný text odpovídá ve všech dvojicích, kandidátní klíč je klíčem skutečným.

Docela překvapením bylo zjištění, že všechny kandidátní klíče jsou použitelné a dávají stejný výsledek - z čeho se dá usoudit jenom to, že na hodnotě K1 nezáleží, jenom záleží na vztahu K2 a K3 vůči K1:

```
$ ./crack.php
000011101 Victory.
001111100 Victory.
010000111 Victory.
011100110 Victory.
100101001 Victory.
101001000 Victory.
110110011 Victory.
111010010 Victory.
```

Závěry

- slabý článek algoritmu je použití jenom tří kroků - protože právě tato jednoduchost nám umožnila odhalit další věci, které pomohly k řešení (i když numerickému).
- odolnost použitím více cyklů by mohla být zvýšena za podmínky, že sub-klíče nebudou na sobě závislé jako v uvedeném příkladě (zamezí se nejspíš mému prostému útoku silou).
- návrhy pro zlepšení - větší počet kroků, ale hlavně provádět substituci tak, aby mapovala více vstupních hodnot na jednu výstupní (u normálního použití se SBOX nepoužívá reverzně)

Příklad 3 - RSA 32

Třída

V tomto příkladu bylo úkolem implementovat základní variantu algoritmu RSA s veřejným exponentem $e=3$. Implementaci jsem provedl v jazyce *PHP* pomocí použití knihovny *BCMath* [BC] která umožňuje práci s číslem libovolné velikosti. Třída pro RSA obsahuje následovní metody (podrobněji budou popsány dále):

```
class RSA {
    function __construct( $key )
    public function putString( $input )
    public function getString( $input )
    public function doBlock( $x )
    // static function modularExponentiation( $a, $b, $n )
    static function modularInversion( $x, $y )
    static function MillerRabin( $n, $t = 8 )
    static function getPrime( $b = 15 )
    static function makeKeys( $b = 31 )
}
```

Implementace algoritmu RSA sestávala z vyřešení těchto úloh a pod-úloh:

- generování klíče
 - nalezení prvočísla
 - nalezení inverzního prvku
- šifrování a dešifrování

Generování klíčů

Algoritmus

Generování klíčů je sice částečně popsáno v zadání, ale pro úspěšnou implementaci jsem musel šáhnout po podrobnější popis, konkrétněji ten v [DI]. Algoritmus generování je známý, takže zde uvedu jen moji implementaci (funkce pro generování prvočísel a nalezení inverzního prvku jsou popsány dále):

```
static function makeKeys( $b = 31 ) {
    $bhalf = floor($b/2); // bit length
    $e = 3; // choosen Public Exponent
    while(1) {
        while(1) {
            $p = self::getPrime( $bhalf ); // two primes
            $q = self::getPrime( $b - $bhalf );
            if ( $p != $q ) break; // must not be same!
        }
        $n = bcmul( $p, $q ); // their multiplication
        $pml = bcsub( $p, 1 );
        $qml = bcsub( $q, 1 );
        $phi = bcmul( $pml, $qml );
        if ( (bcmod( $pml, 3 )==0) || // GCD( phi, 3 ) must be 1
            (bcmod( $qml, 3 )==0) ) continue;
        // To compute the value for d, use the Extended Euclidean Algorithm
        // to calculate d = e^-1 mod phi (this is known as modular inversion).
        if (($D=self::modularInversion( $phi, $e ))!=0) {
            $d = ($D>0) ? $D : $phi+$D;
            if ($d==1) continue; // not a good combination
            break;
        }
    }
    return array( 'private' => array( $n, $e ) // the order of keys...
                , 'public' => array( $n, $d ) // ... does not really mater
                );
}
```

Generování prvočísel

Pro generování prvočísel jsem použil algoritmus z poznámky v [DI]:

Zvolíme si náhodné číslo velikosti $b/2$ kde b je požadovaná velikost klíče, nastavíme nejnižší bit na jedničku (aby jsme zabezpečili lichost čísla) a taky nastavíme dva nejvyšší bity na jedničky (tohle zabezpečí, že v součinu prvočísel bude na nejvyšším bitu také jednička). Zkontrolujeme jestli je právě vytvořené číslo prvočíslo (pomocí *Miller-Rabinova* testu) a když není, přičteme dvojku otestujeme znova.

Takto jsme vygenerovali první prvočíslo, p . Pro generování druhého, q , postupujeme stejně, ale náhodné číslo bude délky $b-b/2$ (po vynásobení $p.q$ dostaneme tedy b -bitový klíč). V bezpečnějších implementacích je možno namísto inkrementování generovat náhodně číslo znova.

Implementace metody je tedy poměrně jednoduchá (za použití *Miller-Rabinova testu*):

```
static function getPrime( $b = 15 ) {  
  
    $p = rand( 0, ( 1 << $b ) - 1 ); // random of bit length (n/2)  
    $p |= 1; // set low bit  
    $p |= 3 << ($b-2); // set the highest two bits  
  
    while ( !self::MillerRabin($p) // while not a prime  
        $p += 2; // try the next one..  
  
    return $p;  
}
```

Miller-Rabinův test je podrobně popsán v [HAC], a algoritmus vhodný na implementaci si nachází na straně 139, bod 4.26. Po implementaci v PHP vypadá kód následovně (poznámky jsou odvozeny od popisu algoritmu):

```
static function MillerRabin( $n, $t = 8 ) {  
  
    $nml = $n - 1;  
  
    // 1. write n-1 as 2^s.d where s is an integer and d is odd  
    $s = 0;  
    $r = $nml;  
  
    // this is the same as factoring out 2 from n repeatedly  
    while( ($r & 1) == 0 ) {  
        $r = $r >> 1;  
        $s++;  
    }  
  
    // 2. repeat t times  
    while( $t-- > 0 ) {  
  
        // 2.1 pick random integer a, from range < 2, n-2 >  
        $a = rand(2,$n-2);  
  
        // 2.2 compute y= a^r mod n  
        $y = bcpowmod($a,$r,$n);  
  
        // 2.3 if...  
        if ( ($y != 1) && ($y != $nml) ) {  
            $j = 1;  
            while ( ($j<$s) && ($y!=$nml) ) {  
                $y = bcpowmod($y,2,$n);  
                if ($y==1) return false;  
                $j++;  
            }  
            if ($y!=$nml) return false;  
        }  
    }  
  
    // is prime  
    return true;  
}
```

Výpočet inverzního prvku

Nejsložitější částí implementace RSA bylo najít dostatečně vysvětlující popis algoritmu na nalezení inverzního prvku. Nakonec se to povedlo najít v [HAC], strana 608 a 610, body 14.61 a 14.64. Algoritmus je založen na rozšířeném binárním hledání největšího společného dělitele a jeho implementace v mojí třídě je:

```
static function modularInversion( $x, $y ) {

    // 1
    $g = 1;

    // 2 while x and y are both even, do x=x/2, y=y/2, g=2g
    while( ( ( $x & 1 ) == 0 ) &&
           ( ( $y & 1 ) == 0 ) ) {
        $x = $x >> 1;
        $y = $y >> 1;
        $g = $g << 1;
    }

    // 3
    $u = $x; $v = $y;
    $A = 1; $B = 0; $C = 0; $D = 1;

    while(1) {

        // 4 while u is even do
        while( ( $u % 2 ) == 0 ) {
            // 4.1
            $u = intval($u / 2);
            // 4.2
            if ( ( ($A % 2) == 0 ) &&
                 ( ($B % 2) == 0 ) ) {
                $A = intval($A/2);
                $B = intval($B/2);
            } else {
                $A = intval(( $A + $y ) / 2);
                $B = intval(( $B - $x ) / 2);
            }
        }

        // 5 while v is even do
        while( ( $v % 2 ) == 0 ) {
            // 5.1
            $v = intval($v / 2);
            // 5.2
            if ( ( ($C % 2) == 0 ) &&
                 ( ($D % 2) == 0 ) ) {
                $C = intval($C/2);
                $D = intval($D/2);
            } else {
                $C = intval(( $C + $y ) / 2 );
                $D = intval(( $D - $x ) / 2 );
            }
        }

        // 6
        if ( $u >= $v ) {
            $u = $u - $v;
            $A = $A - $C;
            $B = $B - $D;
        } else {
            $v = $v - $u;
            $C = $C - $A;
            $D = $D - $B;
        }

        // 7
        if ( $u == 0 )
            // finish a=C,b=D .. (a,b,gv)
            return $D;

        // else go to step 4 (the while cycle)
    }
}
```

Šifrování a dešifrování

Šifrování i dešifrování se provádí stejně, pomocí jednoduchého vzorce:

$$out \leftarrow in^e \pmod{m}$$

Jeho implementace je taky jednoduchá (původně jsem implementoval taky modulární umocňování dle [BKR], ale nakonec jsem našel funkci který provádí to samé v [BC]):

```
public function doBlock( $x ) {
    //return self::modularExponentiation($x, $this->exp, $this->mul);
    return bcpowmod( $x, $this->exp, $this->mul );
}
```

Pomocí této metody jsme schopni zakódovat jediný blok dat - třeba znak. Po výměně klíče za druhý z dvojice vygenerovaných můžeme provést dešifrování - dostaneme původní znak. Pokud se podíváme na zašifrovanou podobu, zjistíme, že i když byl vstupem 8-mi bitový znak, tak na výstupu je bitů víc - a pokud se budeme snažit výstup oříznout, tak se nám nepovede dešifrovat znak.

Pro šifrování celého textu jsem napsal funkci, která vezme řetězec znaků, konvertuje je na bloky, ty zašifruje a vrátí zašifrované bloky. Taky jsem napsal opačnou funkci která provede dešifrování bloku a rozdělení na znaky. Prakticky jsem mohl použít jako vstup šifrovací funkce 24 bitů, tj. tři znaky, ale musel jsem ukládat celý 32-bitový výstup - takže zašifrovaný text je delší o jednu třetinu:

```
$ ./test.php
Private key is: [n=1551288289,e=3]
Public key is: [n=1551288289,d=1034136067]
Encoding:      RSA - It just works.
Cipher:        05c3c0b848d243f8389f70420ac5c52c3247591049ad22682c7e097b
Cipher (ascii): ...H.C.8.pB...,2GY.I."h,~.
Decoding:      RSA - It just works.
```

Pokoušel jsem se použít i 32bitové vstupní bloky, ale nepovedlo se mi je spolehlivě zašifrovat v každém případě - chyba může být jak v matematických knihovnách, tak ve vlastnosti šifry, která vyžaduje určitou redundanci ve vyšších bitech (v některých textech se také vyskytuje doplňování vstupu náhodnými hodnotami - protože pokud bychom šifrovali jen 8 bitů na 32, tak u dlouhého textu by šlo provést frekvenční analýzu). Ukázka, nepovedeného šifrování 32 na 32 bitů:

```
$ ./test.php
Private key is: [n=3116690977,e=3]
Public key is: [n=3116690977,d=2077719211]
Encoding:      RSA - It just works.
Decoding:      RSA - It just works.

$ ./test.php
Private key is: [n=3456749299,e=3]
Public key is: [n=3456749299,d=2265186593]
Encoding:      RSA - It just works.
Decoding:      <NEUh:[k:l?
```

Efektivita nad typem __int64

Algoritmus známý pod pojmem modulární umocňování implementovaný následovně:

```
static function modularExponentiation( $a, $b, $n ) {
    $d = 1; // initial value
    $i = 32; // max. bit + 1
    while($i-->0) { // if i>0 then dec(i) and go in..
        $bi = ( $b >> $i ) & 1; // i-th bit of b, i is from < 31, 0 >
        $d = ($d*$d) % $n;
        if ($bi) $d = ($d*$a) % $n;
    }
    return $d;
}
```

může pracovat s klíčem dlouhým 32 bitů. Výsledek součinu $d.d$ nebo $d.a$ se vleze do 64 bit jenom když činitele jsou 32bitové, a to se dosáhne jenom při operaci modulo N, kde N je 32-bitové. (úvaha založený na řádku $\$d = (\$d*\$d) \% \n).

Literatura

- [BC] **BCMath Arbitrary Precision Mathematics Functions**
<http://www.php.net/manual/en/ref.bc.php>
- [DI] **RSA Algorithm**
http://www.di-mgt.com.au/rsa_alg.html
- [HAC] **Handbook of Applied Cryptography**
Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone
<http://www.cacr.math.uwaterloo.ca/hac/>
- [BKR] **Příklady asymetrických algoritmů, RSA**
(prezentace k přednáškám předmětu Bezpečnost a kryptografie)
<https://www.fit.vutbr.cz/study/courses/BKR/private/bkr04.pdf>