CHAOCIPHER REVEALED: THE ALGORITHM

Moshe Rubin

ADDRESS: Rechov Shaulson 59/6, Jerusalem 95400 ISRAEL; mosher@mountainvistasoft.com.

ABSTRACT: Chaocipher is a method of encryption invented by John F. Byrne in 1918, who tried unsuccessfully to interest the US Signal Corp and Navy in his system. In 1954, Byrne presented Chaocipher-encrypted messages as a challenge in his autobiography "Silent Years". Although numerous students of cryptanalysis attempted to solve the challenge messages over the years, none succeeded. Chaocipher has been a closely guarded secret known only to a handful of persons. Following fruitful negotiations with the Byrne family during the period 2009-2010, the Chaocipher papers and materials have been donated to the National Cryptologic Museum in Ft. Meade, MD. This paper presents the first full disclosure and description of Byrne's Chaocipher algorithm.

KEYWORDS: Chaocipher, John F. Byrne, cryptanalysis, William F. Friedman, National Cryptologic Museum, Silent Years, Lou Kruh, Cipher Deavours

Introduction

The story of John F. Byrne and his Chaocipher encryption scheme has been told in the open literature. The fascinating and colorful story of Chaocipher, from its invention in 1918, through negotiations with William F. Friedman and others in the US Signal Corps and Navy, up to the present locating of the Chaocipher material, and concluding with its donation to the National Cryptologic Museum in Ft. Meade, MD [5] have been amply described in the numerous references (for an introduction to Chaocipher and its history, see [1] and [8]). The author and other researchers plan on writing future papers examining these fascinating technical and historic areas of research.

The purpose of this paper is to disclose the algorithm underlying the Chaocipher encryption system, as described in the papers of John F. Byrne. The author believes that the disclosure of Chaocipher's algorithm will spur other cryptanalysts to research and examine this engaging system which is, interestingly, a simple yet very difficult cryptographic system to break.

Description of Byrne's Primitive Chaocipher Model

It was previously known that John F. Byrne had blueprints for a Chaocipher machine drawn up back in the 1920's. It is clear today that no such machine was ever constructed. The donated Chaocipher material, however, does contain a primitive Chaocipher model made of cardboard and wooden letters (see figure 1, [6]). This model was reconstructed by Byrne's son, John, and provides us with an approximation of the model used by Byrne to encipher the Exhibits in "Silent Years" [4].

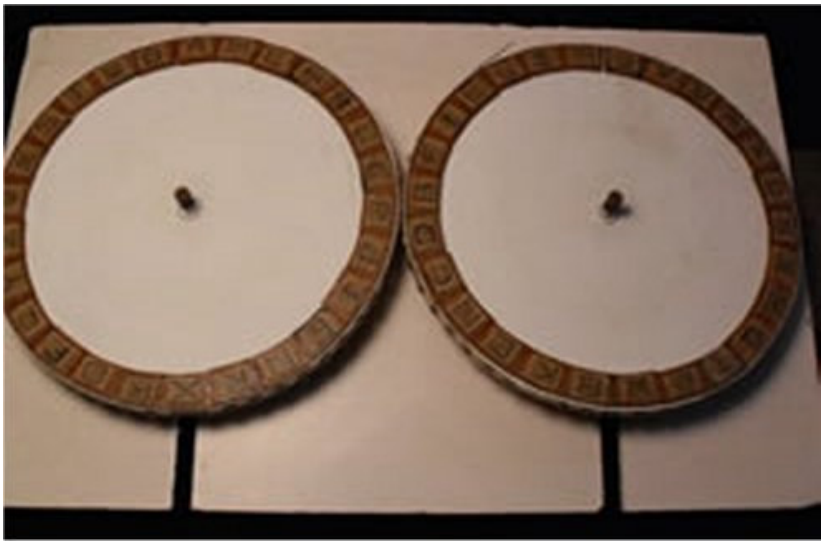

Figure 1. A primitive mechanical model of the Chaocipher device
(photo courtesy of National Cryptologic Museum)

The device consists of two disks, each disk rotating on a spindle. Along the periphery of each disk is a modifiable 26-character alphabet consisting of the letters A to Z in some order. The disks are meant to mesh ("engage") on their periphery so that rotating one of the disks in one direction (i.e., clockwise or counterclockwise) rotates the other disk in the opposite direction with a ratio of 1:1.

The novel principle of the Chaocipher algorithm is each alphabet is slightly permuted each time a letter is enciphered or deciphered. The continuous alphabet permutations result in nonlinear and highly diffused alphabets. The exact method of permuting the alphabets will be described in detail in the next section.

John F. Byrne thought of Chaocipher in mechanical terms, such as "engaging" and "disengaging" the disks to prevent simultaneous rotation of the disks at certain points in the enciphering/deciphering process. The mechanical aspects of Chaocipher will be discussed in a future paper. This paper will focus on the algorithmic aspects of Chaocipher; the model described here is not constrained by mechanical concerns.

The Chaocipher Algorithm Explained

Although Byrne had the physical model in mind when he invented Chaocipher, this paper will use a simplified model that does not require disks. We will represent each disk's alphabet as a 26-character string consisting of the letters A to Z. Figure 2 shows an example of both left and right alphabets, each one being a mixed permutation of the standard alphabet.

```
             +             *
 LEFT (ct): HXUCZVAMDSLKPEFJRIGTWOBNYQ
RIGHT (pt): PTLNBQDEOYSFAVZKGJRIHWXUMC
             --------------------------
  Position: 12345678911111111112222222
                    01234567890123456
```

Figure 2. Left (ct) and right (pt) alphabets in a Chaocipher session

It is important to note that the right alphabet is used for finding the plaintext letter, while the left alphabet is used for finding the corresponding ciphertext letter[1].

Note the two symbols '+' and '*' positioned above the alphabets in figure 2. In Byrne's descriptions these are called the *zenith* and *nadir*, corresponding to the 1st and 14th positions of each alphabet, respectively. These positions will play a major role when permuting each alphabet following each enciphering/deciphering step.

Overview of Chaocipher Process

Given left and right alphabets, with the alphabets aligned relative to their respective *zenith* points, enciphering a plaintext character consists of three stages:

1. Determine the ciphertext letter corresponding to the plaintext letter.
2. Permute the left alphabet.
3. Permute the right alphabet.

These three steps are performed continuously until the plaintext input is exhausted. As an example we will encipher the plaintext letter "A" using the alphabets shown in figure 2.

How to Encipher Plaintext

To encipher a plaintext letter, locate it in the right (pt) alphabet. The letter in the left (ct) alphabet directly above the plaintext letter is the ciphertext letter.

In our example, to encipher the plaintext letter "A", we locate it in the right (pt) alphabet (in the 13th position) and take the corresponding letter directly above it in the left (ct) alphabet, which is a "P". So plaintext "A" is enciphered as ciphertext "P" (see the vertical arrow '↓'):

```
                +           ↓*
 LEFT  (ct): HXUCZVAMDSLKPEFJRIGTWOBNYQ
RIGHT  (pt): PTLNBQDEOYSFAVZKGJRIHWXUMC
             --------------------------
  Position: 12345678911111111112222222
                      01234567890123456
```

Permuting the Alphabets

Now that we know the plaintext letter and its corresponding ciphertext letter, we can proceed to permute the alphabets in preparation for enciphering the next plaintext letter. To stress again, we permute the left and right alphabets with full knowledge of the just-enciphered plain- and ciphertext letters.

Permute the Left Alphabet

Permuting the left alphabet involves the following steps:

1. Shift the entire left alphabet cyclically so the ciphertext letter just enciphered is positioned at the *zenith* (i.e., position 1).
2. Extract the letter found at position *zenith*+1 (i.e., the letter to the right of the *zenith*), taking it out of the alphabet, temporarily leaving an unfilled 'hole'.

[1] It is perfectly logical to alternate between locating the plaintext letter in the right or left alphabet based on some prearranged pattern. As will be shown in a future paper, Byrne used this alternating alphabet method for deriving the starting alphabets.

3. Shift all letters in positions *zenith*+2 up to, and including, the *nadir* (*zenith*+13), moving them cyclically one position to the left.
4. Insert the just-extracted letter into the *nadir* position (i.e., *zenith*+13).

Let's perform the above steps on the left (ct) alphabet using our example. Performing step (1) we shift the entire alphabet to bring the ciphertext letter "P" to the zenith position:

```
                +             *
LEFT (ct): PEFJRIGTWOBNYQHXUCZVAMDSLK
```

Performing step (2), we extract the letter at position *zenith*+1 (i.e., "E") leaving a momentary 'hole'. This leaves the left alphabet looking like this:

```
                +             *
LEFT (ct): P.FJRIGTWOBNYQHXUCZVAMDSLK
```

For step (3) w shift all letters beginning with *zenith*+2 ("F") up to and including the *nadir* ("Q"), moving the sequence ("FJRIGTWOBNYQ") as a complete block one position to the left. The left alphabet now looks like this:

```
                +             *
LEFT (ct): PFJRIGTWOBNYQ.HXUCZVAMDSLK
```

In the final step (4), we insert the extracted letter ("E") back into the alphabet at the *nadir* position:

```
                +             *
LEFT (ct): PFJRIGTWOBNYQEHXUCZVAMDSLK
```

This is the new permuted left alphabet.

Permute the Right Alphabet

Permuting the right alphabet is similar to that of the left alphabet, with small but significant differences. It consists of the following steps:

1. Shift the entire right alphabet cyclically so the plaintext letter just enciphered is positioned at the *zenith*.
2. **Now shift the entire alphabet one more position to the left** (i.e., the leftmost letter moves cyclically to the far right).
3. Extract the letter at position *zenith*+2, taking it out of the alphabet, temporarily leaving an unfilled 'hole'.
4. Shift all letters beginning with *zenith*+3 up to, and including, the *nadir* (*zenith*+13), moving them cyclically one position to the left.
5. Insert the just-extracted letter into the *nadir* position (*zenith*+13).

Let's perform the above steps on the right (pt) alphabet using our example. For step (1) we shift the entire alphabet cyclically to bring the plaintext letter "A" to the *zenith* position:

```
                +             *
RIGHT (pt): AVZKGJRIHWXUMCPTLNBQDEOYSF
```

In step (2) we shift the alphabet one more position to the left, bringing the letter "V" to the *zenith* (don't forget to always do this step for the right-hand (pt) alphabet!):

```
                +             *
RIGHT (pt): VZKGJRIHWXUMCPTLNBQDEOYSFA
```

Next, in step (3), we select the letter two positions to the right of the *zenith* (i.e., "K") and extract it momentarily. This leaves the right-hand (pt) alphabet looking like this:

```
                +             *
RIGHT (pt): VZ.GJRIHWXUMCPTLNBQDEOYSFA
```

For step (4) cyclically shift all the remaining letters following the 'hole' up to, and including, the *nadir* ("GJRIHWXUMCP") one position to the left:

```
                +             *
RIGHT (pt): VZGJRIHWXUMCP.TLNBQDEOYSFA
```

As the last step, step (5), insert the just-extracted letter ("K") back into the alphabet at the *nadir* position:

```
                +             *
RIGHT (pt): VZGJRIHWXUMCPKTLNBQDEOYSFA
```

At this point we have two newly permuted left and right alphabets:

```
                +             *
 LEFT (ct): PFJRIGTWOBNYQEHXUCZVAMDSLK
RIGHT (pt): VZGJRIHWXUMCPKTLNBQDEOYSFA
```

In a real-life case we would be ready to encipher the next plaintext character. The enciphering process is now identical for every plaintext letter until the end of the input (a) find the plaintext letter in the right alphabet, (b) determine the ciphertext letter, and (c) permute the two alphabets.

Enciphering a Longer Plaintext Sequence

Now that you've seen how a plaintext letter is enciphered into its corresponding ciphertext letter, you should prove to yourself that you understand it correctly by performing an exercise: encipher a specific sequence of plaintext letters into ciphertext given the starting alphabets.

The starting alphabets are the same ones presented above in figure 2:

```
                +             *
 LEFT (ct): HXUCZVAMDSLKPEFJRIGTWOBNYQ
RIGHT (pt): PTLNBQDEOYSFAVZKGJRIHWXUMC
```

The plaintext to encipher is:

```
WELLDONEISBETTERTHANWELLSAID
```

You'll know you understand the algorithm if the resulting ciphertext is:

```
OAHQHCNYNXTSZJRRHJBYHQKSOUJY
```

For the record, here are all the alphabets you should have generated in the process:

Left Alphabet (ct)	Right Alphabet (pt)	CT ← PT
HXUCZVAMDSLKPEFJRIGTWOBNYQ	PTLNBQDEOYSFAVZKGJRIHWXUMC	O W
ONYQHXUCZVAMDBSLKPEFJRIGTW	XUCPTLNBQDEOYMSFAVZKGJRIHW	A E
ADBSLKPEFJRIGMTWONYQHXUCZV	OYSFAVZKGJRIHMWXUCPTLNBQDE	H L
HUCZVADBSLKPEXFJRIGMTWONYQ	NBDEOYSFAVZKGQJRIHMWXUCPTL	Q L
QUCZVADBSLKPEHXFJRIGMTWONY	NBEOYSFAVZKGQDJRIHMWXUCPTL	H D
HFJRIGMTWONYQXUCZVADBSLKPE	JRHMWXUCPTLNBIEOYSFAVZKGQD	C O
CVADBSLKPEHFJZRIGMTWONYQXU	YSAVZKGQDJRHMFWXUCPTLNBIEO	N N
NQXUCVADBSLKPYEHFJZRIGMTWO	BIOYSAVZKGQDJERHMFWXUCPTLN	Y E
YHFJZRIGMTWONEQXUCVADBSLKP	RHFWXUCPTLNBIMOYSAVZKGQDJE	N I
NQXUCVADBSLKPEYHFJZRIGMTWO	MOSAVZKGQDJERYHFWXUCPTLNBI	X S
XCVADBSLKPEYHUFJZRIGMTWONQ	AVKGQDJERYHFWZXUCPTLNBIMOS	T B
TONQXCVADBSLKWPEYHUFJZRIGM	IMSAVKGQDJERYOHFWZXUCPTLNB	S E
SKWPEYHUFJZRILGMTONQXCVADB	RYHFWZXUCPTLNOBIMSAVKGQDJE	Z T
ZILGMTONQXCVARDBSKWPEYHUFJ	LNBIMSAVKGQDJOERYHFWZXUCPT	J T
JILGMTONQXCVAZRDBSKWPEYHUF	LNIMSAVKGQDJOBERYHFWZXUCPT	R E
RBSKWPEYHUFJIDLGMTONQXCVAZ	RYFWZXUCPTLNIHMSAVKGQDJOBE	R R
RSKWPEYHUFJIDBLGMTONQXCVAZ	YFZXUCPTLNIHMWSAVKGQDJOBER	H T
HFJIDBLGMTONQUXCVAZRSKWPEY	LNHMWSAVKGQDJIOBERYFZXUCPT	J H
JDBLGMTONQUXCIVAZRSKWPEYHF	MWAVKGQDJIOBESRYFZXUCPTLNH	B A
BGMTONQUXCIVALZRSKWPEYHFJD	VKQDJIOBESRYFGZXUCPTLNHMWA	Y N
YFJDBGMTONQUXHCIVALZRSKWPE	HMAVKQDJIOBESWRYFGZXUCPTLN	H W
HIVALZRSKWPEYCFJDBGMTONQUX	RYGZXUCPTLNHMFAVKQDJIOBESW	Q E
QXHIVALZRSKWPUEYCFJDBGMTON	SWYGZXUCPTLNHRMFAVKQDJIOBE	K L
KPUEYCFJDBGMTWONQXHIVALZRS	NHMFAVKQDJIOBRESWYGZXUCPTL	S L
SPUEYCFJDBGMTKWONQXHIVALZR	NHFAVKQDJIOBRMESWYGZXUCPTL	O S
OQXHIVALZRSPUNEYCFJDBGMTKW	WYZXUCPTLNHFAGVKQDJIOBRMES	U A
UEYCFJDBGMTKWNOQXHIVALZRSP	GVQDJIOBRMESWKYZXUCPTLNHFA	J I
JBGMTKWNOQXHIDVALZRSPUEYCF	OBMESWKYZXUCPRTLNHFAGVQDJI	Y D

Note that the leftmost column in the left alphabet table vertically mirrors the generated ciphertext, while the rightmost column of the right alphabet table corresponds to the plaintext. This property logically stems from the method of generating the alphabets and can serve as a verifying check of your work.

How to Decipher Ciphertext

Deciphering a Chaocipher-encrypted message is identical to the steps used for enciphering. The sole difference is that the decipherer locates the known ciphertext letter in the left (ct) alphabet, with the plaintext letter being the corresponding letter in the right (pt) alphabet. Alphabet permuting is identical in enciphering and deciphering.

Implementing Chaocipher in Software

Although the Chaocipher algorithm is relatively simple once revealed, it is tedious and error-prone if done by hand. It is therefore highly recommendable to implement the Chaocipher algorithm as a software program in the language of your choice. Chaocipher has been implemented in the Perl (see appendix A) and C++ programming languages, and these will hopefully be uploaded in the near future for general usage.

Present and Future Papers

It was decided to concentrate in this paper solely on the algorithmic description of the Chaocipher system. The author deliberately did not describe how to decipher Exhibits 1 and 4 from John F. Byrne's autobiography "Silent Years" to enable would-be decipherers to try their hands at solving them armed only with the knowledge of the system. Anyone interested in doing so can find the challenge messages in computer-readable format [3] on *The Chaocipher Clearing House* [1] web site.

At the present time of writing, exhibits 2 and 3 from "Silent Years", and exhibit 5 from Lou Kruh's and Cipher Deavours's 1990 article in Cryptologia [7], have not yet been deciphered.

Future papers will deal with such topics as deciphering the "Silent Years" exhibits, assessing Chaocipher cryptographic security, the mechanical aspects of Chaocipher as seen by Byrne, cryptanalysis of Chaocipher, Byrne's proposed key distribution scheme, and more.

Conclusion

Numerous cryptanalytic researchers, both professional and amateur, have leveled justified charges against the fact that John F. Byrne violated Kerckhoff's famous principle [2] that "a cryptosystem should be secure even if everything about the system, except the key, is public knowledge". This paper attempts to rectify that valid criticism by revealing the Chaocipher algorithm. Students of cryptanalysis can now try their hands at solving the Chaocipher challenge messages armed with the inner workings of the system.

References

[1] "What is Chaocipher?" The Chaocipher Clearing House web site, http://www.mountainvistasoft.com/chaocipher/what-is-chaocipher.html (last accessed 29 June 2010).

[2] For one of many descriptions of the principle, see Wikipedia: http://en.wikipedia.org/wiki/Kerckhoffs'_principle (last accessed 29 June 2010)

[3] "ASCII versions of all Chaocipher Exhibits", The Chaocipher Clearing House, http://www.mountainvistasoft.com/chaocipher/Chaocipher-ASCII-versions.htm (last accessed 29 June 2010).

[4] Byrne, John F. 1953. Silent Years: An Autobiography with Memoirs of James Joyce and Our Ireland. New York: Farrar, Straus & Young.

[5] The Chaocipher Clearing House, Progress Report #16, http://www.mountainvistasoft.com/chaocipher/chaocipher-016.htm (last accessed 29 June 2010).

[6] "Chaocipher Machine and Papers", web site of the National Cryptologic Museum Foundation, http://www.cryptologicfoundation.org/content/Direct-Museum-Support/recentacquisitions.shtml#Chaocipher (last accessed 29 June 2010)

[7] *Chaocipher Enters the Computer Age When its Method is Disclosed to Cryptologia Editors*; John Byrne, Cipher A. Deavours, Louis Kruh; Cryptologia (1990), Volume 14, Issue 3

[8] "Chaocipher: Analysis And Models", Jeffrey A. Hill (2003, revised 2009), located on The Chaocipher Clearing House web site, http://www.mountainvistasoft.com/chaocipher/chaocipher-009.htm , (last accessed 2 July 2010).

Appendix A: Perl implementation of the Chaocipher Algorithm

```
#-----------------------------------------------------------------------
#
#   ChaocipherSim.pl
#
#      This script simulates the actual, real Chaocipher.
#
#   Written by: Moshe Rubin (June 2010)
#
#-----------------------------------------------------------------------

use strict;
use diagnostics;
use warnings;

# <variables>
my $inFileName = "";
my $outFileName = "";
my $in = "";
my $out = "";
my $ptDiskPattern = "R";   # Default is all plaintext lettern on the right disk
my $diskPatternIndex = 0;
my @traceRangeArray = ();
my %traceRangeHash = ();

# The following are array-based data, offset 0 is "left" and offset 1 is "right".
my @alphabet = ("", "");
my @zenithChar = ("", "");
my @zenithOffset = (-1, -1);

my $DECIPHER_MODE = 1;
my $ENCIPHER_MODE = 2;

my $mode = 0;

parseCommandLine();
validateCommandLine();
displayOptions();
setUp ();

doIt ();

printf "\n";
printf $out;

if ($outFileName ne "")
```

```
{
   open (OUT, ">$outFileName");
   printf OUT $out;
   close (OUT);
}

# . . .

print "\nFinished!\n";

#-----------------------------------------------------------------------
# Subroutines
#-----------------------------------------------------------------------

sub usage
{
   print "\n";
   print "Usage: perl ChaocipherSim.pl -leftalphabet <alphabet>\n";
   print "                             -leftzenith <letter>\n";
   print "                             -rightalphabet <alphabet>\n";
   print "                             -rightzenith <letter>\n";
   print "                             -in <file>\n";
   print "                             -out <file>\n";
   print "                             -encipher\n";
   print "                             -decipher\n";
   print "                             -ptdiskpattern <pattern>\n";
   print "                             -tracerange <from>:<to>\n";
   print "Examples:\n";
   print "\n";
   print "\tperl ... -leftalphabet mpgoihxewctunkflbydavzjrqs \n";
   print "\t         -rightalphabet kwpbudqnoflmistacergvhjxyz -in message.ct.txt\n";
   print "\t         -rightzenith b -leftzenith k -encipher -tracerange 537-560\n";
   print "\t         -ptdiskpattern RLRRLLR\n";
   print "\n";
}

sub parseCommandLine
{
   my $i;
   my $p;

   # Parse command line
   for ($i=0; $i<@ARGV; $i++)
   {
      # Convert parameter to lowercase for comparison
      $p = lc($ARGV[$i]);

    # NOTE: Compare $p with LOWERCASE strings only!

      if ($p eq "-leftalphabet")
```

```
      {
         $alphabet[0] = uc($ARGV[++$i]);
      }
      elsif ($p eq "-rightalphabet")
      {
         $alphabet[1] = uc($ARGV[++$i]);
      }
      elsif ($p eq "-leftzenith")
      {
         $zenithChar[0] = uc($ARGV[++$i]);
      }
      elsif ($p eq "-rightzenith")
      {
         $zenithChar[1] = uc($ARGV[++$i]);
      }
      elsif ($p eq "-in")
      {
         $inFileName = $ARGV[++$i];
      }
      elsif ($p eq "-out")
      {
         $outFileName = $ARGV[++$i];
      }
      elsif ($p eq "-ptdiskpattern")
      {
         $ptDiskPattern = $ARGV[++$i];
      }
      elsif ($p eq "-decipher")
      {
         $mode = $DECIPHER_MODE;
      }
      elsif ($p eq "-encipher")
      {
         $mode = $ENCIPHER_MODE;
      }
      elsif ($p eq "-tracerange")
      {
         push (@traceRangeArray, $ARGV[++$i]);
      }
      elsif   (
               ($p eq "-?")
               ||
               ($p eq "--?")
               ||
               ($p eq "/?")
               ||
               ($p eq "-h")
               ||
               ($p eq "--h")
               ||
```

```
                ($p eq "-help")
                ||
                ($p eq "--help")
            )
      {
         usage();
         exit;
      }
      else
      {
         print ("\nError: Unexpected command line parameter ($ARGV[$i]), aborting\n");
         usage();
         exit();
      }
   }
}

sub validateCommandLine
{
   my $i;
   my $temp;

   if (length($alphabet[0]) != 26)
   {
      print ("\nError: The left starting alphabet must be 26 different characters\n");
      usage();
      exit();
   }

   if (length($alphabet[1]) != 26)
   {
      print ("\nError: The right starting alphabet must be 26 different characters\n");
      usage();
      exit();
   }

   if (length($zenithChar[0]) eq "")
   {
      print ("\nError: You must define the left zenith character\n");
      usage();
      exit();
   }

   if (length($zenithChar[0]) eq "")
   {
      print ("\nError: You must define the right zenith character\n");
      usage();
      exit();
   }
```

```
   if (!(-e $inFileName))
   {
      print ("\nError: The input file \"$inFileName\" does not exist\n");
      usage();
      exit();
   }

   if (($mode != $DECIPHER_MODE) && ($mode != $ENCIPHER_MODE))
   {
      print ("\nError: Please select either -decipher or -encipher\n");
      usage();
      exit();
   }

   for ($i=0; $i<scalar(@traceRangeArray); ++$i)
   {
      my @f = split /\-/, $traceRangeArray[$i];

      if (scalar(@f) != 2)
      {
         print ("\nError: -tracerange %s must for of the form \"from:to\" (e,g., 123:128)\n", $traceRangeArray[$i]);
         usage();
         exit();
      }

      my $from = int ($f[0]);
      my $to   = int ($f[1]);

      if ($from > $to)
      {
         print ("\nError: Incorrect -tracerange option given (%s)\n", $traceRangeArray[$i]);
         usage();
         exit();
      }

      my $j;

      for ($j=$from; $j<=$to; ++$j)
      {
         $traceRangeHash{$j} = 1;
      }
   }

   if (length($ptDiskPattern) == 0)
   {
      print ("\nError: Please provide a PT disk pattern\n");
      usage();
      exit();
   }
```

```
   if (length($ptDiskPattern) > 0)
   {
      # Make sure -ptdiskpattern consists of only 'R' or 'L'
      $temp = uc($ptDiskPattern);

      for ($i=0; $i<length($temp); ++$i)
      {
         my $c = substr ($temp, $i, 1);

         if (($c ne "R") && ($c ne "L"))
         {
            print ("\nError: PT disk pattern can only consist of \"R\"-s or \"L\"-s (e.g. \"RLRRLLR\")\n");
            usage();
            exit();
         }
      }
   }
}

sub displayOptions
{
   printf "\n";
   printf "Session options\n";
   printf "===============\n";
   printf "\tLeft starting alphabet:         $alphabet[0]\n";
   printf "\tRight starting alphabet:        $alphabet[1]\n";
   printf "\tLeft zenith:                    $zenithChar[0]\n";
   printf "\tRight zenith:                   $zenithChar[1]\n";
   printf "\tPT disk pattern:                $ptDiskPattern\n";
   printf "\tInput file:                     $inFileName\n";
   printf "\tOutput file:                    $outFileName\n";
   printf "\tMode:                           %s\n", $mode == $DECIPHER_MODE ? "decipher" : "encipher";
   printf "\n";
}

sub getFile
{
   my ($f) = @_;
   my $text = "";
   my $line;

   open (FILE, "<$f");

   while ($line = <FILE>)
   {
      chomp($line);
      $line =~ s/\s+//g;
      $line = uc($line);

      $text .= $line;
```

```
   }

   close (FILE);

   return $text;
}

sub setUp
{
   $in = getFile ($inFileName);

   printf "\n";
   printf "Input text has %d characters\n", length ($in);
   printf "\n";

   $zenithOffset[0] = index ($alphabet[0], $zenithChar[0]);
   $zenithOffset[1] = index ($alphabet[1], $zenithChar[1]);
}

sub doIt
{
   my $i;

   for ($i=0; $i<length($in); ++$i)
   {
      my $c = uc(substr ($in, $i, 1));
      my $j;
      my $ptDiskIndex = -1;
      my $ctDiskIndex = -1;

      #-------------------------------------------
      #   Determine which disk is PT and which is CT
      #-------------------------------------------
      if (substr($ptDiskPattern, $diskPatternIndex, 1) eq "R")
      {
         $ptDiskIndex = 1;
         $ctDiskIndex = 0;
      }
      else
      {
         $ptDiskIndex = 0;
         $ctDiskIndex = 1;
      }

      # Increment <$diskPatternIndex>, getting ready for the following loop cycle
      $diskPatternIndex = ($diskPatternIndex + 1) % length ($ptDiskPattern);

      if (traceRequested ($i))
      {
         dumpAlphabets (\@alphabet, \@zenithOffset, $ptDiskIndex, $i);
```

```
    }

    #-------------------------------------------
    #   Align wheels to their zeniths
    #-------------------------------------------

    if ($mode == $DECIPHER_MODE)
    {
       # Align cipher wheel with $c, align plain wheel accordingly.
       # Do this by pointing the cipher-zenith to $c, record the offset, then move
       # the plain-zenith the same offset.
       my $targetOffset = index ($alphabet[$ctDiskIndex], $c);
       my $offset = ($targetOffset - $zenithOffset[$ctDiskIndex]) % 26;

       $zenithOffset[$ctDiskIndex] = $targetOffset;
       $zenithOffset[$ptDiskIndex] = ($zenithOffset[$ptDiskIndex] + $offset) % 26;

       #-------------------------------------------
       #   Read off the plain and cipher letter
       #-------------------------------------------

       if (traceRequested ($i))
       {
          printf "(%6d) ct(%s) = pt(%s)\n", $i, $c, substr ($alphabet[$ptDiskIndex], $zenithOffset[$ptDiskIndex], 1);
          printf "\n";
       }

       $out .= substr ($alphabet[$ptDiskIndex], $zenithOffset[$ptDiskIndex], 1);
    }
    else
    {
       # Align plain wheel (plain) with $c, align cipher wheel (cipher) accordingly
       my $targetOffset = index ($alphabet[$ptDiskIndex], $c);
       my $offset = ($targetOffset - $zenithOffset[$ptDiskIndex]) % 26;

       $zenithOffset[$ptDiskIndex] = $targetOffset;
       $zenithOffset[$ctDiskIndex] = ($zenithOffset[$ctDiskIndex] + $offset) % 26;

       #-------------------------------------------
       #   Read off the plain and cipher letter
       #-------------------------------------------

       if (traceRequested ($i))
       {
          printf "(%6d) pt(%s) = ct(%s)\n", $i, $c, substr ($alphabet[$ctDiskIndex], $zenithOffset[$ctDiskIndex], 1);
          printf "\n";
       }

       $out .= substr ($alphabet[$ctDiskIndex], $zenithOffset[$ctDiskIndex], 1);
    }
```

```
# Increment <$zenithOffset[1]> as per Chaocipher instructions
$zenithOffset[1] = ($zenithOffset[1] + 1) % 26;

#-----------------------------------------
#   Permute the wheels
#-----------------------------------------

my $d;
my $insertionOffset;
my @f = ();

#----------
# Permute the left wheel
# Take the letter at zenith+1, move to nadir 13 places past the zenith
#----------

$d = substr ($alphabet[0], ($zenithOffset[0] + 1) % 26, 1);
$insertionOffset = ($zenithOffset[0] + 13) % 26;

# Shift characters at positions ($zenithOffset[0] + 2) through
# ($insertionOffset - 1) one position to the left

#printf "\n";
@f = split //, $alphabet[0];

for ($j=($zenithOffset[0] + 2)%26; $j!=($insertionOffset+1)%26; $j=($j+1) % 26)
{
   # Copy char from $j to ($j-1)%26

   $f[($j-1)%26] = $f[$j];
}

$f[$insertionOffset] = $d;
$alphabet[0] = join "", @f;

#----------
# Permute the right wheel
# Take the letter at zenith+3, move to nadir 13 places past the zenith
#----------

$d = substr ($alphabet[1], ($zenithOffset[1] + 2) % 26, 1);
$insertionOffset = ($zenithOffset[1] + 13) % 26;

# Shift characters at positions ($zenithOffset[1] + 3) through
# ($insertionOffset - 1) one position to the left

#printf "\n";
@f = split //, $alphabet[1];
```

```
      for ($j=($zenithOffset[1] + 3)%26; $j!=($insertionOffset+1)%26; $j=($j+1) % 26)
      {
         # Copy char from $j to ($j-1)%26

         $f[($j-1)%26] = $f[$j];
      }

      $f[$insertionOffset] = $d;

      $alphabet[1] = join "", @f;
   }

   if (traceRequested ($i))
   {
      dumpAlphabets (\@alphabet, \@zenithOffset, -1, $i);
   }
}

sub traceRequested
{
   my ($index) = @_;

   return defined ($traceRangeHash{$index});
}

sub dumpAlphabets
{
   my ($refAlphabet, $refZenithOffset, $ptDiskIndex, $i) = @_;

   printf "(%6d) leftAlphabet:  %s (zenith: %s)\n", $i, $refAlphabet->[0], substr ($refAlphabet->[0], $refZenithOffset->[0], 1);
   printf "(%6d) rightAlphabet: %s (zenith: %s)\n", $i, $refAlphabet->[1], substr ($refAlphabet->[1], $refZenithOffset->[1], 1);

   if ($ptDiskIndex >= 0)
   {
      printf "(%6d) Plain disk is the %s disk\n", $i, $ptDiskIndex == 1 ? "RIGHT" : "LEFT";
   }

   print "\n";
}
```