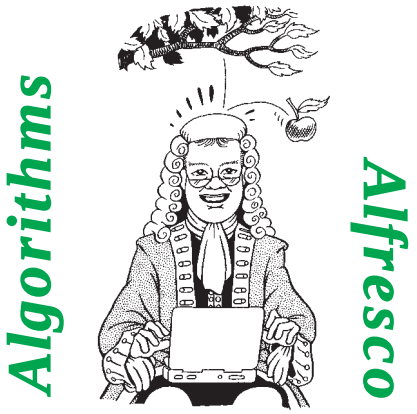
Private Eyes

Lock the doors, draw the curtains: we talk about encryption



by Julian Bucknall

In December 1999's *Algorithms Alfresco* column, I showed how to write a rudimentary spell checker. To aid in this endeavor, I provided a word list: a fairly complete alphabetic list of English words originally compiled for Scrabble players. Originally, I also had the idea of discussing another use for this word list: cracking passwords. However, it turned out that the December article was big enough already, and so I put it to one side.

Then one of my readers asked me whether I'd considered writing an article or two on encryption algorithms. He wanted a Delphi perspective on some of the standard algorithms, and also a back-to-basics introduction to encryption. Okay...

And then came the clincher: *Our Esteemed Editor*, just prior to his wedding at the end of last year, sent me an email, part of which was a suggestion to 'do' encryption in my *Algorithms Alfresco* column. I don't know about you, but when the man with the money 'suggests' something, I tend to listen! And, anyway, why was he thinking about encryption just before his wedding? Mmm...

So, without further ado, encryption.

Adult Education

Before we start in earnest, it will help if I present a small glossary of cryptographic terms. Then we'll be on the same page when we start delving into the algorithms proper.

The whole purpose of cryptography is to enable person A (usually called Alice in the literature) to send a message to person B (who's referred to as Bob). Furthermore, Alice must send this message to Bob with the anticipation that person E (the *eavesdropper* or enemy, usually called Eve) can and will intercept the message somehow. The requirement of cryptography is to alter the message using some algorithm in order to make it extremely difficult for Eve to first of all read the message, and second to substitute another message that Bob would assume to be a real message from Alice. The algorithm that translates the message is known as a *cipher*, or *encryption algorithm*.

The original message that Alice writes is called the *plaintext*. When she applies the encryption algorithm, it will produce another, encrypted, message called the *ciphertext*. Bob's job, when he receives the ciphertext, is to apply the inverse encryption algorithm, called the *decryption algorithm*, to get at the original plaintext. It will come as no surprise that the encryption and decryption algorithms are very closely related, and in fact generally are just known as the encryption algorithm.

Modern algorithms are generally well defined, well known and well studied. To provide the actual security, the algorithms use a *key* (or sometimes, *password*). This key is usually a large binary number and must remain secret between Alice and Bob. Indeed, I'm sure you've heard of 40-bit or 56-bit encryption methods such as the *Data Encryption Standard* (DES), where the 'bit value' is merely the length of the key in bits. In standard DES, for example, the length of the key is 56 bits long, or 7 bytes. To encrypt a message, Alice plugs the key into the encryption algorithm, and to decrypt, Bob plugs the *same* key into the decryption algorithm. These encryption algorithms are known as *symmetric algorithms*. The security of the system is determined by the secrecy of the key: once the key is known by Eve, the system is useless to Alice and Bob. Note that the security of the system is *not* determined by the secrecy of the algorithm. This is a common fallacy. If a system uses a secret algorithm, it usually means that it is easy to crack. In this day and age, with the plethora of public domain and commercial algorithms, it is simply not worth it to use an encryption algorithm that is secret.

Public-key algorithms work in a different way entirely. These algorithms use *two* keys: a public key and a private key. With this system, Bob publishes to the world his public key. When Alice wants to send him a message, she encrypts the plaintext with this public key and sends the ciphertext. When Bob receives the message, he decrypts it with his own personal private key. The most famous program for doing this type of encryption is Pretty Good Privacy (PGP).

Sometimes public-key algorithms work in the opposite fashion, Alice encrypting a message with her private key and Bob decrypting it with Alice's public key. These types of algorithms are usually known as *digital signatures*: the premise being that if Bob can decrypt a message with Alice's public key, the message can only have come from Alice. The message has no secret content whatsoever. Like real signatures, digital signatures are appended to 'documents' and, through a simple application of the public key, Bob can verify that the document is really from Alice. Documents in this context could mean text, or programs, or ActiveX components, or whatever you like. At TurboPower, for example, we

digitally sign the packages for our products to show that they have come from us. We use a company called VeriSign to hold our public key and using a Windows program you can automatically check the provenance of the packages.

In all this discussion, we've been mostly talking about Alice or Bob. What about Eve? What's she going to do to try and read Alice's ciphertext? Here we assume that Eve knows the encryption algorithm used by Alice and Bob, but she doesn't know the key. She would like to crack a message such that she gets to deduce the key and then read all further ciphertext messages with impunity. Each attempt to crack a message is known as an *attack* and the easiest attack to understand is the *brute-force attack*. With this attack, she'll just try key after key after key, until she eventually recovers the plaintext. Here, the longer the key, the longer it will take her. Indeed, the relationship is an exponential one. For an 8-bit key, Eve would have to try up to 2^8 or 256 different keys. For a 16-bit key, twice as long, that's 65,536 different keys. For the DES algorithm, seven times as long as an 8-bit key, that's $7*10^{16}$ different keys. If she could check a billion keys per second it would take her up to 2.28 years to find the key. If Eve knew that Alice and Bob were in the habit of using 7 ASCII characters for their DES key, she wouldn't have to try nearly as many keys to crack a message and indeed could crack it in about 8 seconds at the same rate.

If she's worth her salt, she'll employ a *cryptanalyst* to try and crack the message. *Cryptanalysis* is the science (or art?) of analyzing encryption algorithms together with some ciphertext or plaintext or both in order to identify the key used to encrypt the plaintext.

Guessing Games

Classically (in other words, in times prior to computers), encryption algorithms were very simple. In general, they always acted on letters of the alphabet, generating ciphertext that was also comprised of alphabetic letters. There are two main types of classic encryption algorithms: substitution ciphers and transposition ciphers.

A *substitution cipher* is the simplest and oldest encryption algorithm. Each character in the plaintext is substituted by another character to produce the ciphertext. The most famous of these is known as the *Caesar cipher*. With this algorithm, each character is replaced by the one *n* letters further along the alphabet (with a wraparound at the end, obviously). Thus, if *n* were 3, the cipher Julius Caesar was rumored to use, A is replaced by D, B by E, X by A, Y by B, and so on. Decryption is simple: substitute each character in the ciphertext with the one *n* letters before. For example, you can easily decrypt FDHVDU as CAESAR, knowing that *n* were 3. Indeed, even if you didn't know the value of *n*, the cipher is simplicity to break: there are only 25 possible values of *n* (*n*=0 makes no sense as an encryption algorithm!), and writing a cracker program to break the Caesar cipher is trivial. Prior to the computer age, the easiest way to crack a Caesar cipher was to write the ciphertext down, and then to extend the alphabet down from each ciphertext letter:

```
F D H V D U
-----------
G E I W E V
H F J X F W
I G K Y G X
. . . . . .
B Z D R Z Q
C A E S A R
```

Eventually, you'd get a line that made sense; this would be the plaintext.

Another example of a Caesar cipher is found on UNIX systems, especially for messages on newsgroups. ROT13 is a Caesar cipher with *n*=13, and is easy to use in that a message encrypted with ROT13 is decrypted by applying ROT13 to the ciphertext. Consequently, its main use is not for encrypting messages securely, but for temporarily hiding information so that casual viewers are not upset by it (for example, *risqué* jokes) or that they would only like to see after they'd made a choice to do so (for example, spoilers for an adventure game or movie).

Listing 1 shows code that implements any Caesar cipher: the `N` parameter defines how many letters the algorithm is to advance through the alphabet to encrypt each character in the plaintext. To decrypt, the routine internally executes the same code, but forces `N` to `26-N`. So, for example, if you were to encrypt with `N=3`, the routine would decrypt with `N=23`. ROT13 is simple to implement: `N=13`.

The Caesar cipher is an example of a more generalized *mono-alphabetic cipher*. Here we are still substituting each plaintext letter

➤ *Listing 1: The Caesar cipher.*

```
procedure AACaesarCipher(aEncrypt : boolean; N : integer; aInStream : TStream;
  aOutStream : TStream);
var
  BytesRead : longint;
  i         : integer;
  Ch        : byte;
  Buf       : array [0..255] of byte;
begin
  {force N in range 0..25}
  N := N mod 26;
  if (N < 0) then
    inc(N, 26);
  if not aEncrypt then
    N := 26 - N;
  {read through the input stream in blocks, encrypt the block, and
   write it to the output stream--only convert A-Z and a-z}
  BytesRead := aInStream.Read(Buf, sizeof(Buf));
  while (BytesRead > 0) do begin
    for i := 0 to pred(BytesRead) do begin
      Ch := Buf[i];
      if ((ord('A') <= Ch) and (Ch <= ord('Z'))) then
        Buf[i] := ((Ch - ord('A') + N) mod 26) + ord('A')
      else if ((ord('a') <= Ch) and (Ch <= ord('z'))) then
        Buf[i] := ((Ch - ord('a') + N) mod 26) + ord('a')
    end;
    aOutStream.Write(Buf, BytesRead);
    BytesRead := aInStream.Read(Buf, sizeof(Buf));
  end;
end;
```

to the same ciphertext letter, but now we don't rotate the alphabet to do so, we just randomly generate the translation table. All monoalphabetic ciphers are very insecure since they can all be attacked by the letter frequency method: calculating the frequencies of ciphertext letters. In the English language, the letters E, T, A, I and N predominate, so the first thing to try would be translating the most frequent ciphertext letter to an E and see what we got. We continue along in this vein, experimenting and trying to recognize individual words until we have fully decoded the ciphertext. It must be noted that if we can detect individual words in the ciphertext, it makes cracking much easier so, in general, the ciphertext is sent without spaces or punctuation.

Unguarded Minute

A variation of this is the *Vigenère cipher*. This algorithm is a *polyalphabetic substitution cipher* where we use several simple substitution ciphers sequentially. This cipher also requires a password or key, a word that will be used to determine the substitutions required. Suppose Alice wanted to encrypt the string 'This is a message' with the password 'SECRET'. First she uppercases everything, and then removes all non-alphabetic characters. She writes down this compressed message with the password (repeated, if necessary) underneath:

```
THISISAMESSAGE
SECRETSECRETSE
```

The substitution depends on the current letter of the password line. The first letter of the password line is S. She writes the alphabet down, and then underneath the alphabet rotated so that the S appears under the A:

```
ABCDEF...TUVWXYZ
STUVWX...LMNOPQR
```

She can now encode the T in the plaintext as an L. The next letter of the password line is E, so she generates a new conversion table in the same manner:

```
ABCDEFGH...VWXYZ
EFGHIJKL...ZABCD
```

From this we see that the next ciphertext letter is L again. She continues like this until the entire plaintext is encoded. Bob, when he receives the message, performs the same kind of method to decrypt the message. To make it easier for both Alice and Bob, Vigenère ciphers usually use a 26*26 table of characters with each line having the alphabet shifted left by 1 from the previous one. Figure 1 shows this table. Remember this is in the days prior to computers: although this system is named the Vigenère cypher, it was first devised by Giovan Batista Belaso in 1553; Blaise de Vigenère proposed a slight modification where the plaintext follows the password, instead of the password endlessly being repeated.

To encode, you find the password letter along the left and the plaintext letter along the top and the point where the designated row and column cross is the ciphertext letter. To decode, find the password letter along the side, follow the row until you reach the ciphertext letter and then look up the column to the top to find the plaintext letter.

And Eve? What does she do? She knows that Alice and Bob are using a Vigenère cipher, so she sets to work. The first thing to note is that she must have a ciphertext that is much longer than the key, because the first thing she tries to do is work out the length of the key that was used for encryption. To do this she takes the ciphertext and compares it with the ciphertext that has been rotated by 1, 2, 3, etc, letters. She then counts the number of coincident letters at each position for each rotation. Because of the fact that the plaintext is written in English, and certain plaintext letters are more common than others, she'll find that rotations that are a multiple of the key length will have more coincidences than rotations that are not. This will give a good indication of the length of the key. From this she has a good basis for cracking the rest of the message and finding out the key.

Suppose she finds that the key length was 6. This means that letters 1, 7, 13, etc, were all encoded with the same Caesar cipher. Letters 2, 8, 14, etc, were encoded with another Caesar cipher, and so on. Using a table of the frequencies of the letters in the English language, she can now make assumptions about the ciphertext letters by matching frequencies of ciphertext against English language frequencies. Without too much trouble, she should be able to work out each Caesar cipher, and hence the password, and hence the plaintext. Obviously for this to have some chance of success, the ciphertext must be several times the length of the key or password used to produce it.

Listing 2 is an implementation of the Vigenère cipher. There are four parameters to the routine: whether to encrypt or decrypt, the key, the plaintext as a stream, and

➤ *Figure 1: A complete Vigenère substitution table.*

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
BCDEFGHIJKLMNOPQRSTUVWXYZA
CDEFGHIJKLMNOPQRSTUVWXYZAB
DEFGHIJKLMNOPQRSTUVWXYZABC
EFGHIJKLMNOPQRSTUVWXYZABCD
FGHIJKLMNOPQRSTUVWXYZABCDE
GHIJKLMNOPQRSTUVWXYZABCDEF
HIJKLMNOPQRSTUVWXYZABCDEFG
IJKLMNOPQRSTUVWXYZABCDEFGH
JKLMNOPQRSTUVWXYZABCDEFGHI
KLMNOPQRSTUVWXYZABCDEFGHIJ
LMNOPQRSTUVWXYZABCDEFGHIJK
MNOPQRSTUVWXYZABCDEFGHIJKL
NOPQRSTUVWXYZABCDEFGHIJKLM
OPQRSTUVWXYZABCDEFGHIJKLMN
PQRSTUVWXYZABCDEFGHIJKLMNO
QRSTUVWXYZABCDEFGHIJKLMNOP
RSTUVWXYZABCDEFGHIJKLMNOPQ
STUVWXYZABCDEFGHIJKLMNOPQR
TUVWXYZABCDEFGHIJKLMNOPQRS
UVWXYZABCDEFGHIJKLMNOPQRST
VWXYZABCDEFGHIJKLMNOPQRSTU
WXYZABCDEFGHIJKLMNOPQRSTUV
XYZABCDEFGHIJKLMNOPQRSTUVW
YZABCDEFGHIJKLMNOPQRSTUVWX
ZABCDEFGHIJKLMNOPQRSTUVWXY
```

```
procedure AAVigenereCipher(aEncrypt : boolean;
  aKey : string; aInStream : TStream; aOutStream : TStream);
var
  BytesRead : longint;
  i, j      : integer;
  Ch        : byte;
  Buf       : array [0..255] of byte;
  OutBuf    : array [0..255] of byte;
  KeyValues : array [0..255] of byte;
  KeyLen    : integer;
  KeyInx    : integer;
begin
  {the Vigenere cipher is for uppercase alphabetic letters
    only; in calculating the key values assume the key is in
    such a state}
  KeyLen := 0;
  for i := 1 to length(aKey) do
    if ('a' <= aKey[i]) and (aKey[i] <= 'z') then begin
      KeyValues[KeyLen] := ord(aKey[i]) - ord('a');
      inc(KeyLen);
    end
    else if ('A' <= aKey[i]) and (aKey[i] <= 'Z') then begin
      KeyValues[KeyLen] := ord(aKey[i]) - ord('A');
      inc(KeyLen);
    end;
  if not aEncrypt then
    for i := 0 to pred(KeyLen) do
      KeyValues[i] := 26 - KeyValues[i];
  {read through the input stream in blocks, encrypt the
    block, and write it to the output stream--only convert
    and write A-Z and a-z}
  KeyInx := 0;
  BytesRead := aInStream.Read(Buf, sizeof(Buf));
  j := 0;
  while (BytesRead > 0) do begin
    for i := 0 to pred(BytesRead) do begin
      Ch := Buf[i];
      if ((ord('A') <= Ch) and (Ch <= ord('Z'))) then begin
        OutBuf[j] := ((Ch - ord('A') + KeyValues[KeyInx])
          mod 26) + ord('A');
        inc(j);
        KeyInx := (KeyInx + 1) mod KeyLen;
      end
      else if ((ord('a') <= Ch) and (Ch <= ord('z'))) then
        begin
        OutBuf[j] := ((Ch - ord('a') + KeyValues[KeyInx])
          mod 26) + ord('A');
        inc(j);
        KeyInx := (KeyInx + 1) mod KeyLen;
      end;
    end;
    aOutStream.Write(OutBuf, j);
    BytesRead := aInStream.Read(Buf, sizeof(Buf));
    j := 0;
  end;
end;
```

➤ *Listing 2: The Vigenère cipher.*

finally the ciphertext, also as a stream.

I Can't Go For That

There is a variant of the Vigenère cipher that is used throughout the computer industry. It is easy to implement, and not particularly difficult to break either. I'm talking about the XOR cipher.

Alice and Bob agree on a key. This key doesn't have to be purely alphabetic characters, in fact it's better if it isn't, and the longer this key is, the better. To encrypt plaintext, Alice starts off with the first byte of the message and the first byte of the key. She XORs them together, and outputs the resulting byte as the first byte of the ciphertext. She advances one character, and does the same. When she runs out of bytes in the key, she starts over from the beginning of the key again. To decrypt, Bob does exactly the same, and the plaintext will be recovered, since XORing a byte twice with the same value results in the original byte.

Listing 3 has this simple XOR cipher. Again we pass in a key and two streams, one stream for the plaintext and one for the cipher text.

To crack this cipher, Eve proceeds in the same manner as for the Vigenère cipher. She first attempts to find coincidences between the ciphertext and the ciphertext shifted by various displacements. The shifts that are multiples of the key length will have more coincidences than those that are not. From this we can deduce the key length. Now we take the ciphertext and XOR it with itself shifted along by the key length. This will basically XOR the text with itself and removes the key entirely. From this we can then start applying some letter distributions to try and crack the XORed message.

This type of cipher is used in various applications all over the place. Possibly the most egregious use of the XOR cipher is with a Windows CE system: you set a password that is used to protect your palmtop. When you switch the device on, you are prompted for the password. Simple enough. Anyway, ActiveSync, the program you run to allow you to synchronize the data on your palmtop with your desktop, uses the same password.

```
procedure AAXORCipher(aKey : PByteArray; aKeyLen : integer; aInStream : TStream;
  aOutStream : TStream);
var
  Buf    : array [0..1023] of byte;
  KeyInx : integer;
  i      : integer;
  BytesRead : longint;
begin
  {read through the input stream in blocks, XOR the block with the key
   and write it to the output stream}
  if (aKey = nil) or (aKeyLen = 0) then
    raise Exception.Create('Cannot encrypt with XOR: the key is missing');
  KeyInx := 0;
  BytesRead := aInStream.Read(Buf, sizeof(Buf));
  while (BytesRead > 0) do begin
    for i := 0 to pred(BytesRead) do begin
      Buf[i] := Buf[i] xor aKey^[KeyInx];
      KeyInx := (KeyInx + 1) mod aKeyLen;
    end;
    aOutStream.Write(Buf, BytesRead);
    BytesRead := aInStream.Read(Buf, sizeof(Buf));
  end;
end;
```

➤ *Listing 3: The simple XOR cipher.*

When you start ActiveSync, it prompts you for your Windows CE password. You are given the option to save the password on your desktop machine so that you don't have to supply it every time you want to synchronize your machines. It saves the password, encrypted, in your registry. However, the encryption it uses is a simple XOR cipher with 'susageP' as the key. Why this key? Well, the code name for Windows CE was 'Pegasus' and if you reverse it, you get 'susageP' (see www.cegadgets.com/artsusageP.htm for the full sorry story).

Actually, having severely lambasted the XOR cipher, there is a way in which it can be used to produce an unbreakable cipher. This unbreakable cipher is known as the *one-time pad*. Classically, the

one-time pad was a Vigenère cipher, but with a key that was totally random and the same length as the plaintext. What happens is this. Alice and Bob get together and create a pad of pages, with each page having a sequence of letters on it generated randomly. The pad is duplicated with Alice taking one and Bob taking the other. Now when Alice wants to send Bob a message she uses the standard Vigenère cipher, but with each letter of the plaintext being married to each letter in sequence from the top page of the pad. She continues to encrypt the plaintext, using as many pages of the pad as are required. Once done, she sends the ciphertext and then destroys the pages she has used. Bob uses his pad in the same manner to decrypt the ciphertext, and when he's done he destroys the same pages. Since the key is totally random and, furthermore, is never used again (hence, *one-time* pad), there is no way Eve could get a handle on the original plaintext. She can't use letter frequencies since the same letter will be randomly encrypted with a different key letter every time. There's no attack to try and deduce the key length, since the length of the key is the same as the length of the plaintext. Totally unbreakable.

In the XOR case, the same thing happens. A very large set of random bytes is generated and Alice and Bob both get a copy. Alice encodes her plaintext by extracting out the same number of bytes from her random number table as there are bytes in the plaintext and then XORs them together. The ciphertext is sent, and Alice destroys the random bytes she has used. Bob gets the ciphertext, and uses the same number of random bytes as there are bytes in the ciphertext, and destroys the used random bytes in the same manner as Alice. Again, Eve has no chance to decrypt the ciphertext for the same reasons as in the classic case.

Mind you, if Alice and Bob created their random bytes by seeding the Delphi random number generator and then calling it to generate random bytes, Eve has a chance again. Since the entire sequence of random bytes generated by this method is deterministic (in other words, if Eve started with the same seed she'd get the same random bytes), she has a chink that she can lever open. All she has to do is to try each seed in sequence, until the ciphertext is cracked. If you like, the one-time pad suddenly becomes an encryption method with a 32-bit key and, if Eve has a fast enough machine, she would be able to crack the ciphertext fairly quickly. Nevertheless, such randomized encryption algorithms are fairly popular and there is a whole subset of random number generators that are designed to be cryptographically secure.

Your Imagination

After all this exposition on substitution ciphers, we should take a look at transposition ciphers, the other classical method for encrypting plaintext. A *transposition cipher* is one where the letters in the plaintext are not converted in any way; it is just their *order* that is jumbled up. The simplest of these ciphers is the columnar transposition cipher. Write the plaintext on squared paper, with one letter per box. Limit the width of the squared paper to a certain number of cells. Read off the ciphertext a column at a time. So, for example, if the plaintext was 'ONCE MORE UNTO THE BREACH DEAR FRIENDS,' and our squared paper was 10 boxes wide, we'd get

```
ONCEMOREUN
TOTHEBREAC
HDEARFRIEN
DS
```

By reading the letters vertically, we get the ciphertext 'OTHDNODS...' To decrypt this message we need to know the width of the squared paper. Count the letters in the ciphertext and divide this by the width to give you the number of letters to write vertically and so the ability to decode the ciphertext.

In World War I, the Germans devised the ADFGVX cipher, a mixture between a transposition cipher and a substitution cipher. It was a sophisticated encryption algorithm for its day. It's also a 'wordy' algorithm, since every letter in plaintext is converted into two letters in the final ciphertext.

Create a 6*6 square, naming each row and column after the letters A, D, F, G, V, X, as in Figure 2. Randomly put each letter of the alphabet into this square, together with the ten digits (36 characters in all). The first stage is a substitution cipher: we replace each letter in the plaintext with the row and column identifier according to where the letter was found. For example, using Figure 2, the plaintext letter J would be replaced by DF, since J is at the junction of row D and column F.

Now we have to decide on a password. Let's use the word SECRET. Discard any repeated letters in the password, so we have SECRT. Write the intermediary ciphertext out (the one that just consists of ADFGVX letter pairs) under the word SECRT as in the standard columnar transposition cipher. Here we will have only five columns, of course. Now we read off the letters in the columns, not from the first to the last, but according to the position of the column header letter in the alphabet. So, for SECRT, we read the C column first, then the E column, the R column, the S column and, finally, the T column. The result is a hodge- podge of As, Ds, Fs, Gs, Vs, and Xs. To decrypt we'd need to know the password and also the substitution table.

To demonstrate the algorithm, Alice decides to send the message

➤ *Figure 2: An example substitution table for the ADFGVX cipher.*

	A	D	F	G	V	X
A	U	H	N	A	X	O
D	2	B	J	V	D	4
F	P	T	3	K	5	G
G	E	1	I	S	9	7
V	O	F	8	C	W	Y
X	Z	Q	M	R	6	L

'THEYAREONTOYOU'. She performs the first substitution cipher, replacing the T with FD, the H with AD, and so on. She places the resulting letter pairs into a 5-column transposition table under the word SECRT, as in Figure 3. She then reads the C column first: AXGFV, the E column: DVGAX, and so on, to make the ciphertext AXGFVDVGAX...

Bob can reconstruct the transposition table because he knows the password, and since he also knows the substitution table he can work out the original plaintext. And poor Eve? Well, there doesn't seem to be any rhyme or reason to the collection of these 6 letters, repeated and shuffled seemingly at random. But, surprisingly, this cipher was actually broken using

➤ *Listing 4: The ADFGVX cipher.*

```
procedure AAADFGVXCipher(aEncrypt : boolean; aKey : string;
  const aSubstTable: TaaADFGVXTable; aInStream : TStream;
  aOutStream : TStream);
const
  ADFGVX : array [0..5] of char = 'ADFGVX';
var
  BytesRead : longint;
  i, j      : integer;
  Ch        : char;
  Buf       : array [0..255] of char;
  DblBuf    : array [0..511] of char;
  CleanKey  : string;
  KeyLen    : integer;
  KeyInx    : integer;
  PosCh     : integer;
  MinInx    : integer;
  ColLen    : integer;
  Row, Col  : integer;
  SubstTextSize : longint;
  InStreamSize  : longint;
  MemStream     : TMemoryStream;
begin
  {clean up the key so that it consists only of unique
    uppercase characters}
  CleanKey := '';
  for i := 1 to length(aKey) do begin
    Ch := upcase(aKey[i]);
    if ('A' <= Ch) and (Ch <= 'Z') then
      if (Pos(Ch, CleanKey) = 0) then
        CleanKey := CleanKey + Ch;
  end;
  {===ENCRYPTION===}
  if aEncrypt then begin
    {read the entire input stream, converting into
      letterpairs into a temporary stream}
    MemStream := TMemoryStream.Create;
    try
      BytesRead := aInStream.Read(Buf, sizeof(Buf));
      while (BytesRead > 0) do begin
        j := 0;
        for i := 0 to pred(BytesRead) do begin
          Ch := upcase(Buf[i]);
          if (('A' <= Ch) and (Ch <= 'Z')) or
             (('0' <= Ch) and (Ch <= '9')) then begin
            PosCh := Pos(Ch, aSubstTable) - 1;
            DblBuf[j] := ADFGVX[PosCh div 6];
            DblBuf[j+1] := ADFGVX[PosCh mod 6];
            inc(j, 2);
          end;
        end;
        MemStream.Write(DblBuf, j);
        BytesRead := aInStream.Read(Buf, sizeof(Buf));
      end;
      {now read the letters in each column according to the
        order of the letters in the cleaned key}
      KeyLen := length(CleanKey);
      for KeyInx := 1 to KeyLen do begin
        {find the smallest letter in the key, this is the
          column we'll be reading next}
        MinInx := 1;
        for i := 2 to KeyLen do
          if (CleanKey[i] < CleanKey[MinInx]) then
            MinInx := i;
        CleanKey[MinInx] := #127; {so we don't see it again}
        dec(MinInx); {it's easier with a 0-based number}
        {starting off with the MinInx'th letter in the
          temporary stream, copy it and every KeyLen'th
          letter after that to the output stream}
        SubstTextSize := MemStream.Size;
        PosCh := MinInx;
        j := 0;
        while (PosCh < SubstTextSize) do begin
          MemStream.Position := PosCh;
          inc(PosCh, KeyLen);
          MemStream.Read(DblBuf[j], 1);
          inc(j);
          if (j = sizeof(DblBuf)) then begin
            aOutStream.Write(DblBuf, sizeof(DblBuf));
            j := 0;
          end;
        end;
        if (j > 0) then
          aOutStream.Write(DblBuf, j);
      end;
    finally
      MemStream.Free;
    end;
  end else begin
  {===DECRYPTION===}
    {first create memory stream to use as an intermediary,
     set its size to the size of the input stream}
    InStreamSize := aInStream.Size;
    MemStream := TMemoryStream.Create;
    try
      MemStream.SetSize(InStreamSize);
      {now read the letters in each column according to the
        order of the letters in the cleaned key}
      KeyLen := length(CleanKey);
      for KeyInx := 1 to KeyLen do begin
        {find the smallest letter in the key, this is the
          column we'll be reading next}
        MinInx := 1;
        for i := 2 to KeyLen do
          if (CleanKey[i] < CleanKey[MinInx]) then
            MinInx := i;
        CleanKey[MinInx] := #127; {so we don't see it again}
        dec(MinInx); {it's easier with a 0-based number}
        {calculate the length of this column}
        ColLen := InStreamSize div KeyLen;
        if ((InStreamSize-(ColLen*KeyLen)) > MinInx) then
          inc(ColLen);
        {copy the column from the input stream to the
          temporary stream, starting off by copying to the
          MinInx'th letter in the temporary stream, and
          every KeyLen'th letter after that; we stop at the
          end of the column}
        PosCh := MinInx;
        while (ColLen > 0) do begin
          if (ColLen > sizeof(Buf)) then
            BytesRead := aInStream.Read(Buf, sizeof(Buf))
          else
            BytesRead := aInStream.Read(Buf, ColLen);
          dec(ColLen, BytesRead);
          for i := 0 to pred(BytesRead) do begin
            MemStream.Position := PosCh;
            inc(PosCh, KeyLen);
            MemStream.Write(Buf[i], 1);
          end;
        end;
      end;
      {now read the temporary stream as letter pairs,
        converting them into the original characters for the
        output stream}
      MemStream.Position := 0;
      BytesRead := MemStream.Read(DblBuf, sizeof(DblBuf));
      j := 0;
      Row := 0;
      Col := 0;
      while (BytesRead > 0) do begin
        for i := 0 to pred(BytesRead) do begin
          if not Odd(i) then begin
            case DblBuf[i] of
              'A' : Row := 0;
              'D' : Row := 1;
              'F' : Row := 2;
              'G' : Row := 3;
              'V' : Row := 4;
              'X' : Row := 5;
            end;
            case DblBuf[i+1] of
              'A' : Col := 0;
              'D' : Col := 1;
              'F' : Col := 2;
              'G' : Col := 3;
              'V' : Col := 4;
              'X' : Col := 5;
            end;
            Buf[j] := aSubstTable[Row * 6 + Col + 1];
            inc(j);
            if (j = sizeof(Buf)) then begin
              aOutStream.Write(Buf, sizeof(Buf));
              j := 0;
            end;
          end;
        end;
        if (j > 0) then
          aOutStream.Write(Buf, j);
        BytesRead := MemStream.Read(DblBuf, sizeof(DblBuf));
      end;
    finally
      MemStream.Free;
    end;
  end;
end;
```

the fact that the two stages are totally disconnected from each other, and that the transposition password is not used in the substitution.

The code? Well, it's very messy, as you can imagine. Listing 4 shows the full ADFGVX cipher code and, as you can see, we can't share much code between encryption and decryption as in the previous three ciphers. This month's disk also has a routine to generate a random shuffling of the letters and digits for the substitution table.

Looking For A Good Sign

As a final note in this article on classic ciphers, I would like to mention the ENIGMA machine. This was the encryption device used by the German Army and Navy in the Second World War. I'd quickly like to discuss it for two reasons. Firstly, it was a very sophisticated encryption machine. And, secondly, breaking its encryption fell to a team of mathematicians at Bletchley Park in Buckinghamshire, including one of the fathers of modern computing, Alan Turing. The ENIGMA machine was essentially a *rotor machine*, using three rotors. A *rotor* is a wheel that is wired to perform a monoalphabetic substitution. The wheel was divided into 26 segments, one for each letter, and was designed in such a way that each segment on the face was wired to a different segment on the reverse. If each segment on the reverse was wired to a light, applying electrical current to a segment on the face would cause the light for the encrypted letter to light up. The ENIGMA machine connected its three rotors in series, so each rotor applied a substitution. To ensure that the same substitution was not used over and over, the rotors would be rotated by a number of segments after a letter was encrypted. Furthermore, the ENIGMA machine used the concept of a *reflecting rotor* that caused the circuit to be reflected through the three rotors again, backwards, doubly encrypting each plaintext letter. There was a keyboard by which the plaintext could be typed for encryption (or the ciphertext for decryption), the action of pressing a key would perform the rotations, and the resulting ciphertext letter was read off a set of 26 lights when power was applied through the rotors. To make it even harder to crack, the initial position of the rotors could be defined prior to encrypting or decrypting the message. There was also a plugboard that remapped pairs of letters prior to the encryption. All in all, a fearsome encryption machine.

Breaking the ENIGMA encryption relied on complex mathematics from the most brilliant mathematicians in England, and also quite a bit of luck. The Allies had managed to obtain an ENIGMA machine and some of the code books (these books detailed which rotors were to be used on which days, initial positions of the rotors, and so on), and they had a stroke of luck in that the operators disliked changing the rotors (apparently, a tedious process), so one of the encryption possibilities was negated. Nevertheless, the British team, including Alan Turing, managed to break the encryption by building a machine called The Bombe, which helped analyze the ciphertext. Considering the importance of Turing's work during the War, his subsequent treatment by the British Government in 1952, when he was arrested for a homosexual liaison and stripped of his security clearance, was nothing short of scandalous, and probably led to his death by suicide in 1954.

Crime Pays

This month's disk has the full code for all the ciphers, together with a small driver/test program that encrypts and decrypts with each encryption algorithm. You get to supply the text to be encrypted and you can see the encrypted and decrypted versions of that text.

```
S E C R T
---------
F D A D G
A V X A G
X G G A A
X A F F D
A X V X A
```

➤ *Figure 3: Encoding with the ADFGVX cipher.*

I hope you've enjoyed this small foray into classic encryption algorithms. In the near future, I plan to continue this topic by looking at some of the famous modern computer encryption algorithms such as DES and RSA. Until then, HAPPYPROGRAMMING.

References

Applied Cryptography
by Bruce Schneier.
www.turing.org.uk
for information on Alan Turing.

Julian Bucknall is busy writing the book for the new millennium, listening to early 80s music in the hall, eating porridge. He can be contacted via email at julianb@turbopower.com. The code that accompanies this article is freeware and can be used as-is in your own applications.