

The Humber Cipher Project

L D Howe

No current Affiliation (Retired)

Original affiliation: AEA Technology Culham Laboratory,
Abingdon, Oxon., OX14 3DB, UK

Keywords: cipher, symmetric, universal, security

Abstract

Humber is a symmetric, private key, virtual cipher machine written in FORTRAN. It uses a quadruple key register, double encryption algorithm. The keys are generated by an associated remote multiple tandem pseudo random number generator (PRNG). Its bitwise exclusive OR encryption provides a universal approach that allows it to encrypt and decrypt files with any file name, of any type and of any size, subject to a maximum length of 30 Mbytes. The ease of operating Humber is described. This paper includes a detailed description of the encryption and decryption processes. It also describes the key generation process. It is assumed that Humber is in the public domain and the security of the cipher depends entirely on the security of the keys, which has been assessed in this paper. The performance has been measured and reported here.

Introduction

The Humber project was developed as a symmetric, private key, stream cipher program, together with an associated key generation program. It is a two stage, quadruple key algorithm where the keys are implemented as array elements in the form of an index register and a shift register for each stage of encryption. Encryption and decryption use a bitwise exclusive OR (XOR) process so any file in any format can be encrypted and decrypted, subject to a maximum file size of 30 Mbytes. Any file name less than 200 characters long may be used for files to be encrypted.

The 8192 bit key is generated by a multiple tandem pseudo-random number generator (MTNG) with a time-based disconnection between the initial seed value and the eight seeds used to generate the key. The sender and receiver must use an identical copy of the keys. The key generator output is a FORTRAN source code file that is compiled into the HUMBER executable file. The executable file is distributed to a network users and the source code key file is then destroyed. The security of the ciphers allows for the fact that the cipher machine is publicly available and depends only on the security of the keys. Therefore the keys must be kept secret: otherwise anyone in possession of them will have the ability to decrypt any intercepted encryption. Humber is distributed as an executable application and anyone in possession of the same distribution version will be able to decrypt any message encrypted using that version. However, if a different distribution has been used for encryption, the messages cannot be decrypted. This allows a small group of people to create an exclusive message network. If the key becomes compromised because one of the machines falls into the wrong hands, a new distribution can be arranged with a new embedded key. A complete copy of the source code is available, except for the keys which are specific to a particular distribution and must be kept secret.

Each encryption is assigned a random 32 bit rotation index that indicates how the key file should be set up at the start of decryption and the program also calculates a random 32 bit hash to change the initial register settings. The combination of key values, rotation index and hash is unique to a single encryption. Humber calculates a random 32 bit message number that is used as part of the file name for the encrypted file, in conjunction with the system date. As part of the encryption process, Humber writes a header for each encryption indicating the rotation index, and the hash. Humber can either produce an encrypted character stream for normal operation or the characters can be output (and input) as hexadecimal pairs for the purpose of plaintext challenges or to allow transparency of operation. The original file name is appended to the header and the whole header is encrypted using the original key values. The header also indicates whether or not the encrypted file has been converted to hexadecimal format. This is added to the header, unencrypted, to avoid the possibility of its use as a crib in decrypting messages. The encrypted message is saved as using the created file name consisting of the system date and the random message number. The encrypted message header and new file name ensure that there is no indication of the type of file originally encrypted.

Development

The Humber project was originally conceived as a software implementation of ALVIS (BID610), implemented in an Excel spreadsheet as a feasibility study. I first encountered ALVIS in 1963 at the Cipher School in Catterick. Some fifty years later, after reading Hugh Sebag-Montefiori's "Enigma", I determined to investigate the feasibility of recreating an ALVIS-like shift register algorithm for encryption and decryption. The first attempt was a simple spreadsheet embodiment with a 7 character key in the form of shift register, using formulae and look-up references. This was quickly augmented by a Visual Basic (VB) program written in the spreadsheet, which used a data file, input file and output file. It also used a genuine XOR function for the encryption. The result of encryption for each character was a 7 bit word in the form of ASCII ones and zeros. However, because the shift register was recharged with the encrypted character, it was relatively simple to decrypt the messages without knowing the key.

The next development was to include a 256 character index register to encode each character. This implementation was known as Lanchester, a Coventry company which, like ALVIS, made non-mainstream cars. The output was changed to 8 bit ASCII characters encoded as hexadecimal pairs. The index register was used to encrypt and decrypt each character, whilst the 19 character shift register was used to point to the required element of the index register. However, the character added to the highest element of the shift register was still the encrypted character, so it was still possible to use cribs to identify elements of the plaintext. The message number was included as a header and used to rotate the registers prior to encryption and decryption. In order to overcome the problem with cribs, the character added to the highest element of the shift register was later changed to the plaintext character prior to encryption. This completely disconnected the encrypted character from the shift register contents.

At this stage, Lanchester was re-written as a FORTRAN application with a 256 character index register and a 199 character shift register. This meant it used a 3630 bit key in the form of the initial register settings. Another FORTRAN program was written as a secure PRNG, providing a suite of key files randomly shuffled so that it was not possible to identify the order in which they were created. The key generation process was developed over a period of time to ensure that the multiple seed values were genuinely random. Humber is based on Lanchester, Humber being another Coventry Car manufacturer of high specification cars. Humber introduced the use of a stream output and double encryption.

Principle of Encryption

The encryption and decryption processes are virtually mirror images of each other. Each byte of the source file is converted to its ASCII value and encrypted (or decrypted) sequentially. All the numbers and operations within Humber are represented by 8-bit values (decimal 0 through 255) and the index registers have 256 elements, representing all the possible values found in the shift registers. Each shift register also has 256 elements.

Each character is encrypted using a value in the index register. The particular element of the index register is selected by the first element of the shift register. The shift register is shifted right and a new value is calculated for its highest element. Every character is encrypted or decrypted twice using two identical processes, each with its own pair of registers. The decryption process is a reverse of the encryption process, as shown in Figure 1.

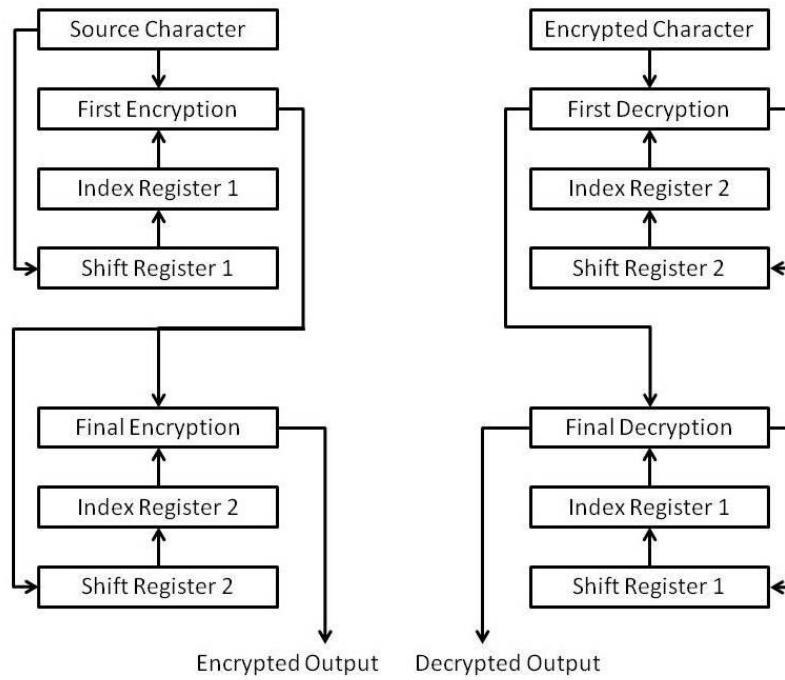


Figure1: Encryption and Decryption

File Structure

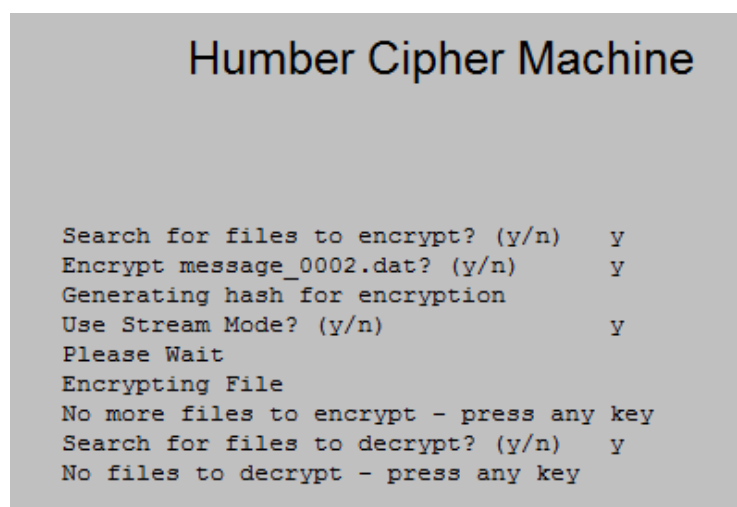
Humber creates a journal file. This file shows the time of every Humber transaction and indicates whether it is an encryption or decryption. It then shows the name of the encrypted file alongside the name of the plain language file. This allows the operator to pair up the files if any need to be re-encrypted or re-decrypted. The journal file is essential in this capacity because there is no correlation between the original file name and the encrypted file name. This file also allows the timing of messages to be tracked. The journal file resides in same directory as the executable and is named "Journal.dat".

Humber creates six subdirectories, if not already present. Assuming the name of the installation directory is "humber" the six subdirectories would be:

1. "humber/messages". This contains all the files for encryption. Once a file has been created, it must be placed in this directory before it can be encrypted.
2. "humber/crypto". This contains all the encrypted files ready for transmission. All the file names are created automatically by Humber during the encryption process and each consists of the system date followed by a random, 8 character, hexadecimal number".
3. "humber/encrypted". All messages are moved automatically from the messages directory to this directory when they have been encrypted.
4. "humber/cripin". This contains all the enciphered messages for decryption. These will have been created by a Humber application and transmitted as encrypted files. Once received, messages for decryption must be placed in this directory.
5. "humber/plaintext". This contains all the decrypted plain text files with original file name, which are automatically added to this directory by Humber.
6. "humber/decrypted". All messages are moved automatically from the cripin directory to this directory when they have been decrypted.

The Operator's Perspective

On starting the Humber cipher machine, the operator receives the message "Search for files to encrypt? (y/n)". If "n" is selected the control will move on to decryption. If "y" is selected, Humber then creates a list of all the files in the "messages" folder. If there are any files in the "messages" folder, the first is offered for encryption via the prompt "Encrypt *filename* (y/n)", where *filename* is the name of the first unmatched file. If the operator accepts the file, the prompt "Generating hash for encryption" appears followed by the prompt "Use Stream Mode? (y/n)". Rejection of Stream Mode results in Hex Mode being used for the cipher. Hex Mode will cause the output to appear as hexadecimal pair representations of the output stream. Once the choice has been made, encryption proceeds immediately. A typical operator's control window is shown in Figure 2.



```
Humber Cipher Machine

Search for files to encrypt? (y/n)    y
Encrypt message_0002.dat? (y/n)     y
Generating hash for encryption
Use Stream Mode? (y/n)              y
Please Wait
Encrypting File
No more files to encrypt - press any key
Search for files to decrypt? (y/n)   y
No files to decrypt - press any key
```

Figure 2: A typical operator's control window

Each message file is offered in turn. If the operator rejects a file, the next one is offered until there are no more unencrypted files. The same procedure is followed for decryption, this time using the files in the "cripin" folder.

The Encryption Process

A PRNG is used to assemble the hash function, calculate the rotation index and form a message number. The PRNG begins with an initial value (IV) and is allowed to cycle for a fixed number of operations, calculating a new 64-bit random number (X_i) each time. The number of operations in each cycle depends on the value of X_i and the number of milliseconds indicated by the system clock at the beginning of the cycle. At the end of the each cycle, the polarity of the random number is reversed switching to a new part of the X_n sequence that can only be predicted from the value of X_i at the end of the cycle. After ten such cycles, the system clock is checked to see if it indicates a multiple of 10ms. If not, the cycles continue until the end of a cycle when a multiple of 10 ms is indicated. The value of X_i is then used to calculate the hash function. The process continues in a similar manner to generate rotation index and the message number. Each of the above numbers is a 32 bit number that is converted to an 8 byte hexadecimal value.

Before the encryption process can begin the file names must be identified. The input file name is captured for use in the message header, so that the original file name can be recovered during decryption. The name of the encrypted file is a composite of the system date and the message number created by the random process, separated by a period and followed by ".dat". The embedded keys are then assigned to the various registers. The keys are embedded as hexadecimal pairs, each of which has to be converted to an integer value before being assigned to a register element. There are 1024 hexadecimal pairs corresponding to 256 elements in each of the four registers. As each element contains an 8 bit number, the key set is essentially an 8192 bit key.

Next, the message data must be read from the message file. This is achieved by reading the data in 64kB blocks. As each block is read it is inserted into the message data stream, which takes the form of a single string (character) variable. Currently, the size limit for the message is 3×10^7 characters, which is about 7,500 pages of A4 script or 5 million words. It is possible to increase the maximum size of the message by more than an order of magnitude, should the need arise. The data reading process is illustrated in Figure 3.

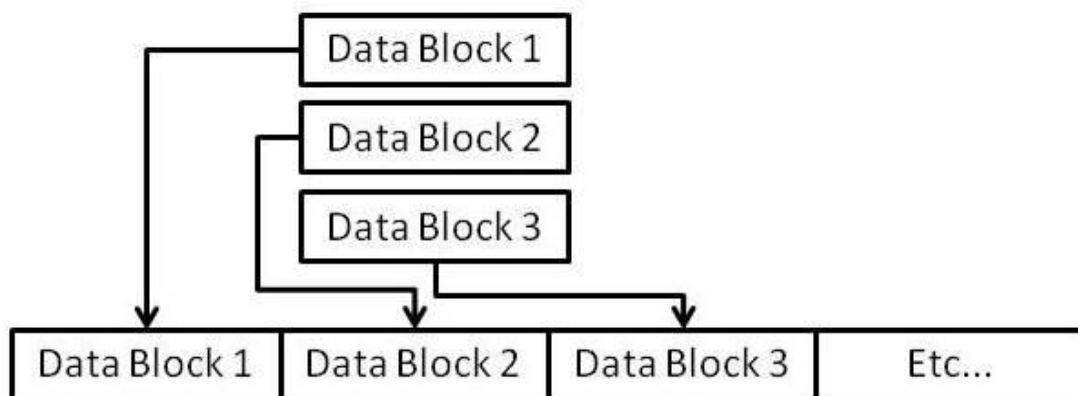


Figure 3: Arrangement for reading message data in blocks

Once the message data has been captured in the form of a single data stream, the message header is created for the encrypted file. The header contains 4 parts. The first two characters, transmitted unencrypted, are a hexadecimal pair representing the message mode, either “.” for stream mode or “z” for hexadecimal mode, converted respectively to 2E and 7A. The following two parts comprise the hash and the rotation index generated by the random process. The final part is the original name of the file to be encrypted. These are assembled as a plain language header and then encrypted, as described by Equations (2) and (3) below, using the initial register setting. The encrypted header is then added to the first two characters as a series of hexadecimal pairs. The final character of the message header is an unencrypted colon (:), which is used to indicate the end of the header. A typical message header is shown in Figure 4.

2EFF3AFA1457CF05BC9722C4A86D2633ED4BDDDB5FD0E3EEEC5AB0A2A6A5482:

Figure 4: Typical message header

After header generation, Humber then rotates the registers from the positions at the end of header encryption. In the case of all four registers, the rotation used is ROTATE RIGHT CIRCULAR. The rotations produce the message settings for the message being encrypted. This means that no matter how many times the same message is encrypted, it will always produce a different cipher stream. The rotation index is treated as four hexadecimal pairs, each pair indicating the number of rotations for one register. To achieve this, the hexadecimal pair is converted to a decimal value (n). The number of rotations (n) for each register lies in the range $0 \leq n < 255$. The rotation process is shown in Figure 5.

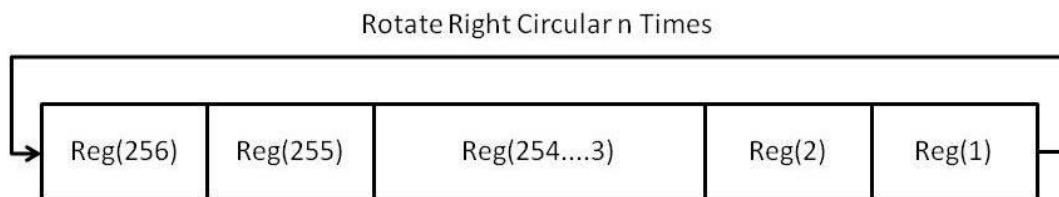


Figure 5: Rotation of the registers to find message settings

Once the registers have been set to the correct value, the hash function is split into four in the same way and each part modifies the values of every element of one register via a bitwise XOR function:

$$\text{Reg}(i) = \text{Reg}(i) \text{ XOR } n \tag{1}$$

The two stages of encryption are identical. For each stage of the 2-stage encryption the following notation is used (the subscript i corresponds to the i^{th} character from the plain language message):

C_i = the ASCII value of the i^{th} enciphered character;
 R_i = the value of the index register element used for encrypting or decrypting the i^{th} character
 $R(j)$ = the value of the j^{th} element of the index register;
 $S(j)$ = the j^{th} element of the shift register;
 S_i = the value of $S(1)$ for the i^{th} character;
 P_i = the ASCII value of the i^{th} plaintext character.

For the first stage of encryption, the first encryption is represented by C_i , while for the second stage, the first encryption is represented by P_i . The encryption process begins by using a bitwise XOR function to combine the ASCII value of each plain language character in turn with the contents of the appropriate element of the index register, indicated by the value of S_i :

$$C_i = P_i \text{ XOR } R(S_i) \quad (2)$$

This produces the enciphered character, C_i . The XOR function is also used to combine the same plain language character with the contents of S_i , the result of which is added to the shift register as the next value of $S(256)_{i+1}$, after a SHIFT RIGHT LOGICAL operation on the shift register. The value of S_i used for the encryption is overwritten and cannot be recovered:

$$S(256)_{i+1} = P_i \text{ XOR } S_i \quad (3)$$

In practise, if there were a large number of repeated identical characters in the plain language text, a pattern would emerge in the output. The pattern could be used to identify the keys, if the repeating character can be discovered. Therefore an additional step is incorporated whereby a PRNG generates a random number between 0 and 255 to be combined with the result of Equation (3)

$$S(256)_{i+1} = P_i \text{ XOR } S_i \text{ XOR } R_i \quad (4)$$

where R_i is the random number. Thus S_i is a number used once only (nonce). The random number generator seed does not need to be secure because its only function is to mask the effect of repeated characters.

The purpose of the shift register process is to ensure there is no traceable connection between $S(256)_{i+1}$ and the enciphered character C_i . The whole encryption process is illustrated by Figure 7.

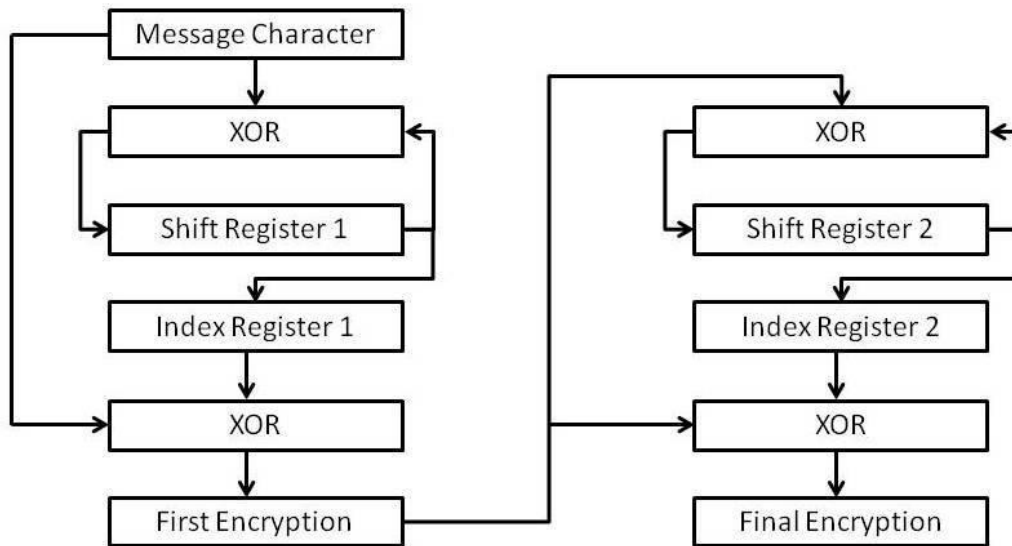


Figure 6: Details of the encryption process

As each character is encrypted, it is added to the output stream. In the case of the default mode, the final encryption is converted to a character representation of the ASCII code and added to the output stream. In Hex Mode, the number is written as a 2-Byte hexadecimal number, followed by a space (ASCII 32).

Once encryption is complete, the output file is opened to write the encrypted file. The encrypted file has the same filename as the original message with the suffix “.humber” added to indicate that it has been encrypted by Humber. First, the file header is written as a distinct record. Then the output stream is written as a single record, after which the file is closed, ready for transmission.

The Decryption Process

Although the decryption process is essentially a mirror image of encryption, there are notable differences. Before the message data is read from the message file, the file header must be recovered. This must be followed by a 2 byte data read to allow for the end of the record (line feed and carriage return). The length of the message header (n) must be calculated before decryption can proceed. After the message header has been read, the encrypted data is read in 64k byte blocks as for encryption. As each block is read it is inserted into the message data stream, which takes the form of a single string (character) variable. However, the first block is limited to less than 65,536 bytes, because the message header accounts for the first n bytes of the block. Otherwise the process is the same as for encryption, shown in Figure 3.

After the data has been read, the header must be analysed. Before the header data can be analysed, the embedded keys must be assigned to the various registers, using an identical process to that used for reading the key file during encryption.

The header must be converted from hex pairs to decimal numbers. The first two characters of the header are used to set the decryption mode to either “Stream” or “Hex”. The encrypted part of the header is decrypted in the same manner as for the message decryption (described by Equations (5) and (6) below). It is then split into the three elements as described for encryption. The first group of eight characters is used to determine the hash. Characters 9 to 16 are captured as the rotation index for the decryption. The remainder of the header from Character 17 is captured as the original file name of the file before encryption.

Rotation of the registers occurs exactly as shown in Figure 5. Once the registers have been set to the correct value, the hash function is split into four and each part modifies the values of one register via an XOR function, as during encryption.

The decryption process has to be carried out in reverse to the encryption process. In Hex Mode, the characters are recovered from the input stream three at a time. The first two are read as a hexadecimal, single byte pair and the third, a space (ASCII 32) is neglected. This recovers the original, encrypted ASCII code. In Stream mode, the characters are recovered one at a time and converted to the ASCII equivalent. The notation here is the same as that used to describe the encryption process except that for the first stage of decryption, the first decryption is represented by P_i , while, for the second stage, the first decryption is represented by C_i .

The decryption process begins by using the XOR function to combine the ASCII value of each cipher text character in turn with the contents of the appropriate element of the index register, indicated by the value of S_i :

$$P_i = C_i \text{ XOR } R(S_i) \quad (5)$$

This produces the deciphered character, P_i . The XOR function is also used to combine the deciphered character with the contents of S_i , the result of which is added to the shift register as the next value of $S(256)_{i+1}$, after a SHIFT RIGHT LOGICAL operation on the shift register. The value of S_i used for the decryption is overwritten and cannot be recovered.

$$S(256)_{i+1} = P_i \text{ XOR } S_i \quad (6)$$

As with encryption, a random number between 0 and 255 is combined with the result of Equation (6)

$$S(256)_{i+1} = P_i \text{ XOR } S_i \text{ XOR } R_i \quad (7)$$

where R_i is the random number. Because the PRNG is reset to the original seed at the beginning of each encryption and decryption, this will reproduce the same value of $S(256)_{i+1}$ as that produced during encryption. The whole encryption process is illustrated by Figure 7.

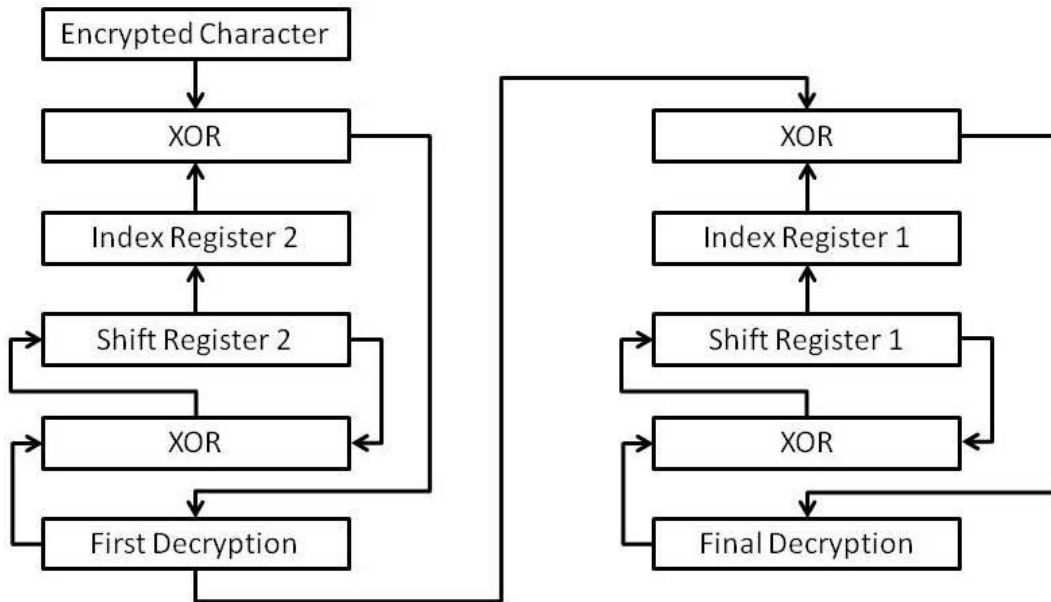


Figure 7: The decryption process

As each character is decrypted, it is added to the output stream. Once the decryption is complete, the output file is opened to write the output stream. The output file has the same file name as the original message. Because all the original control characters, such as carriage returns and linefeeds were enciphered along with the original text, the output stream can be written as a single record, after which the file is closed. The plaintext file will be identical to the original message file.

Key Generation

A dedicated random number generator was developed to generate the secret keys for Humber. The keys are generated by a multiple tandem pseudo-random number generator (MTNG) which comprises four tandem PRNGs. Each tandem PRNG has an array of 256 elements. The array has to be primed before the tandem PRNG can begin to generate number sequences. A PRNG requires an initial random number (the seed) and all subsequent random numbers are produced by applying a predictable mathematical operation on the current value of the random number:

$$X_{i+1} = P(X_i) \quad (8)$$

X_0 is the seed, X_i is the i th value of the random number, X_{i+1} its next value and P represents the mathematical operation.

In order to identify the correct value of the seed, any attacker would need to test the assumed value against encrypted messages. Clearly, if there is only one possible value of seed, it will immediately be discovered and the test will be a formality. Increasing the number of possibilities, increases the time it will take to test all possible values. Let us assume that a modern supercomputer with multi-parallel processing can check 2^{32} values of seed in one second. If the seed (and subsequently generated random numbers) were a 32 bit integer, it would take no more than one second to identify the correct value of the seed. However, if it were a 64 bit integer, the same super computer would take about 136 years to check every possible value. Almost all modern portable and transportable devices can manipulate 64 bit integers, so a 64 bit format was chosen as the basis for the seed and random numbers.

It is essential to use a suitable form of $P(X_i)$ that produces a good random distribution and will generate all possible values such that $0 \leq X_i < T$, where T is the basis (e.g. 2^{64}). Hull and Dobell [1] describe a suitable function as

$$X_{i+1} = aX_i + b \pmod{T} \quad (9)$$

In order to produce all possible values without repetition, it is only necessary to satisfy $a = 1 \pmod{4}$ and $b = 1$. The correlation, ρ_s , between X_n and X_{n+s} is given by

$$\rho_s = \frac{1 - 6\frac{b_s}{T}\left(1 - \frac{b_s}{T}\right)}{a_s} + e \quad (10)$$

where

$$\begin{aligned} a_s &= a^s \pmod{T} \\ b_s &= (1 + a + a^2 + \dots + a^{s-1})b \\ |e| &< \frac{a_s}{T} \end{aligned}$$

If a is chosen so that $a \approx \sqrt{T}$ the correlation $\rho_1 \approx T^{-1/2}$. By using $a = 2^n + m$ the only operations required are one shift left of n places plus two additions. This provides the fastest method of computing the next value of X_i . If $T = 2^{2n}$ with $n > 1$, then, by using and $a = \sqrt{T} + 1$, a good performance can be expected in response to statistical tests. In this case the next value of X_i is calculated via:

$$X_{i+1} = 4294967317X_i + 1 \quad (11)$$

4294967317 is 21 greater than 2^{32} , which, in turn, is the square root of 2^{64} , the basis for seed generation. The integer multiplication process neglects any overflow values, so X_i is always a valid 64-bit number.

The concept of a tandem PRNG was developed for Monte Carlo simulations of physical processes. It has been tested over many years with a variety of problems [e.g. 2, 3]. A tandem PRNG gives a better statistical spread of random numbers than a simple PRNG and, as such, is well suited to cryptographic purposes. A tandem PRNG requires two random streams from two seeds. Before the tandem PRNG can generate random numbers it must be primed. The first stream of random numbers is used to populate an array variable **A** such that

$$A(i) = X1_i \times 2^{-56} + 128 \quad (12)$$

where $A(i)$ is the i th element of the array and $X1$ is the i th random number from the first stream. The Value of $A(i)$ is always an integer in the range 0 to 255, corresponding to the values required for the register keys. The addition of 128 is required because the processor treats the unsigned integer value of the seed as a signed integer between -2^{63} and $+2^{63} - 1$. The priming process involves only one of the two seeds required to operate the tandem PRNG. Once the array has been primed with all 256 values, the tandem PRNG operates by using one seed to select the array element to be used for the next random number and the second seed to generate a new value to be entered into the array in place of the one that has just been used. Mathematically, the random number, n_i , is given by:

$$n_i = A(j), j = X1_i \times 2^{-56} + 128 \quad (13)$$

The replacement value is given by:

$$A(j) = X2_i \times 2^{-56} + 128 \quad (14)$$

In (13) and (14), $X1$ represents the first random stream and $X2$ the second.

When the MTNG is first started, a single seed stream, X_n , is generated by a simple PRNG from a fixed seed value (IV) and is allowed to cycle for a fixed number of operations, calculating a new 64-bit random number (X_i) every 11ns (about 93 million seed values per second). The number of operations in each cycle depends on the value of X_i and the number of milliseconds indicated by the system clock at the beginning of the cycle. At the end of the each cycle, the polarity of the random number is reversed switching to a new part of the X_n sequence that can only be predicted from the value of X_i at the end of the cycle.

$$X_{i+1} = -X_i \quad (15)$$

The purpose of the inversion is to move X_i onto another section of the X_n chain. This is necessary because even at 93 million new values per second, it would take over 4000 years to generate all 2^{64} possible values, which means that without the reversals, an attacker would only need to check a limited number of possible seed values. After the required number of such cycles, the system clock is checked to see if it indicates a multiple of 10ms. If not, the cycles continue until the end of a cycle when a multiple of 10 ms is indicated.

To create each tandem PRNG seed, the process is repeated until the system clock indicates a multiple of 10 ms. At this point, the current value of the seed, X_i , is captured as the seed. The current value is once again inverted. The whole process is repeated until all eight seeds have been generated. Because the value of X_i is updated every 11ns, it is regarded as incredible that the same chain could be replicated, even by the same MTNG, so the value of the seed can be regarded as truly random.

After eight values have been captured, the MTNG primes the arrays and generates 4 streams, each of 256 random numbers in the range 0 – 255. The 4 streams are used for the initial values assigned to the Humber registers, 1 stream to each of two index registers and two shift registers. Every random process is generated by its own tandem PRNG, using two different random number sequences, both with a unique, non-predictable, initial seed. Each of the four streams is generated by a separate tandem PRNG. Every key value is converted to a hexadecimal pair and the values are saved to a FORTRAN file as 64 streams of 16 hexadecimal pairs for inclusion in the compilation to create an embedded key in the Humber executable.

Security of Encryption

It is assumed that the Humber Cipher algorithm is in the public domain. Therefore, the security of encryption depends entirely on the security of its register keys. Each index register has 2^{2048} possible initial settings and each shift register has 2^{2048} possible initial settings. Therefore the probability of selecting the correct initial settings for Humber as a whole is 1 in 2^{8192} . Providing the confidentiality of the keys remains intact, the most likely attack would be on the PRNG processes. The MTNG requires eight seeds. Each seed has 2^{64} possible values, so the probability of selecting the correct set of 8 initial key generation seeds is 1 in 2^{512} .

Because there are no pre-defined seeds for the PRNG processes, it is not possible to identify them and reproduce the generation of the keys. Once all the seed values have been captured, the keys are generated. At the end of the whole process, the values of the seeds are discarded so they can never be discovered to allow the rollback generation of the key files.

Performance

The maximum file size that can be accommodated by Humber is 30 MB (in Hex mode the maximum plain language file size is 10 MB). The speed of reading input files depends critically on the size of the maximum file size and the block size during the data read operations. If larger files are required to be encrypted, the time taken to read in the data will be increased, unless the block size is also increased.

The performance tests were run on a Dell Inspiron 545 Desktop computer fitted with an Intel Core 2 Quad CPU Q8300 running at 2.5 GHz. The computer had 4 GB RAM with a 64-bit operating system running under Windows 7. Two test files were encrypted and decrypted. Because the files have to be read in blocks and assigned to variables, the read time for large files becomes sufficiently significant to be measured. On the other hand, the output is written as a single stream, so the write time is negligible. File 1 was a binary bit map file “Dump.bmp”. File 2 was a Word file “Humber.docx”. The recorded times for encryption and decryption are shown in Table I:

	File Size	Encryption		Decryption	
		Read Time	Process Time	Read Time	Process Time
File 1	5.12 Mb	4.3 secs	9.7 secs	4.3 secs	9.7 secs
File 2	315 kB	< 0.5 sec	< 0.5 sec	< 0.5 sec	< 0.5 sec

Table 1: Recorded performance data for test files

The recorded times do not include the time taken to carry out the random processes for the hash function, key file ID generation and message setting number, which take approximately 4 seconds in total for any encryption, but are not required for decryption.

Conclusions

Humber is a quadruple key, double encryption cipher system with an associated multiple tandem PRNG. Any file of any file type can be encrypted and decrypted by Humber, providing that the file does not exceed the maximum allowed file length (currently 30 MB, with a theoretical maximum of 4 GBytes). The usability for an operator has been shown to be straightforward. The probability of finding the initial register settings is so low that it may be regarded as impossible, providing the key files remain physically secure. The probability of finding the initial seed values of all eight seed streams is also regarded as incredible. The encryption and decryption times are acceptable.

References

- [1] T E Hull and A R Dobell *Random Number Generators* SIAM Review **Vol4 No. 3** (July 1962)
- [2] L D Howe, D K Ross and A J Allen *Molecular Flow in a Model Pore System in Dynamics in Small Confining Systems* p23 (Eds J M Drake, J Klafter and R Kopelman) **MRS EA-22** 1990
- [3] L D Howe *Studies of Traffic Flow Phenomena Using the VEDENS Computer Code* Physica A **246** (1997)