CONTENTS

CHAPTER

LIST OF FIGURES viii

I. INTRODUCTION 1

A. Motivation 2

B. Approach 4

II. HISTORY OF CRYPTOGRAPHY 6

III. ELEMENTS OF ENCRYPTION 8

A. Substitution 8

1. Monoalphabetic 8

2. Polyalphabetic 10

B. Permutation 11

C. Noise Addition 12

D. Feedback & Chaining 13

1. Plain Text Feedback 15

2. Cipher Text Feedback 15

E. Analog Encryption 15

IV. FACTORS RELATED TO ENCRYPTION 17

A. Change of Language 17

B. Digitization 18

C. Compression 18

D. Multiplexing 19

V. COMPARISON OF SELECTED ALGORITHMS 20

A. One-Time Key Tape 20

B. Linear Shift Register Feedback 21

C. Exponential Encryption 22

D. Knapsack 24

E. Rotor Machines 24

F. Codes 27

G. Galois Field and Hill Cryptosystems 28

H. DES 30

VI. DESIGN CONSIDERATIONS FOR MPJ 34

A. Strength Based on Key 34

B. Usability of Random Keys 34

C. Key Length & Block Size 35

D. Effort Required to Break 35

E. Computational Efficiency 36

F. Communication Channel Efficiency 37

G. No Back Doors or Spare Keys 37

VII. MPJ ENCRYPTION ALGORITHM 38

A. Description 38

1. Overall Structure of MPJ 38

2. Substitution Boxes 40
3. Wire Crossings 40
4. Key Generation 41
B. Implementation in Pascal 45
1. Exceptions from Standard Pascal 45
2. Main Program 47
3. Procedures Encrypt & Decrypt 48
4. Procedures Permute & Ipermute 48
5. Procedures Substitute & Isubst 49
C. Implementation in Hardware 49
D. Strengths & Weaknesses 51
VIII. DATA COMPRESSION 52
A. Purpose 52
B. Linguistic Parsing 55
C. Huffman Coding 57
D. Pascal Programs 57
IX. CONCLUSION 60
REFERENCES 61
A. MPJ in Pascal 66
B. Linguistic Data Compression Programs 83

LIST OF FIGURES

FIGURE

III.B Permutation. 11

III.C. Noise Addition 12

III.D. Block Cipher Modes. 14

III.E. Analog Time & Frequency Encryption 16

V.A. One-time key tape (AKA One-time pad) 21

V.A.1 Typex Rotor Machine 25

V.E.2. Wired Rotors 25

V.H.1. DES Enciphering 30

V.H.2. DES Nonlinear Function 31

V.H.3. DES Internal Key Generation 32

VII.A.1. Overall Structure of MPJ 39

VII.A.3. Wire Crossings 41

BEYOND DES:

DATA COMPRESSION AND THE MPJ ENCRYPTION ALGORITHM

by

Michael Paul Johnson

B. S., University of Colorado at Boulder, 1980

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Electrical Engineering

1989

This thesis for the Master of Science degree by

Michael Paul Johnson

has been approved for the

Department of

Electrical Engineering

by

_____

Mark A. Wickert

_____

Charles E. Fosha, Jr.

_____

Rodger E. Ziemer

Date_____

Johnson, Michael Paul (M. S., Electrical Engineering)

Beyond DES: Data Compression and the MPJ Encryption Algorithm

Thesis directed by Assistant Professor Mark A. Wickert

Many encryption algorithms have come and gone as cryptography, cryptanalysis, and technology have progressed. Today's communication and computer technologies need cryptography to truly secure data in many applications. The demands on the cryptography needed for some commercial applications will exceed the security offered by the National Bureau of Standards Data Encryption Standard (DES) in the near future due to advances in technology, advances in cryptanalysis, and the increasing rewards for breaking such a heavily used algorithm. To meet part of this need, a new block encryption algorithm is proposed. A Pascal program to implement this algorithm is given.

One way to further increase security of encrypted data, as well as to achieve storage and/or transmission economy, is by redundancy reduction prior to encryption. A linguistic approach to redundancy reduction, together with an example computer program to implement it, is given for this purpose.

I. INTRODUCTION

The increasing proliferation of digital communication and computer data base storage has brought with it increasing difficulty of maintaining the privacy of that data. There is only one effective way to protect the privacy of communications sent over such channels as satellites, terrestrial microwave, and cellular telephones. This is by encryption. It is clearly impossible to deny unauthorized access by a determined and knowledgeable interceptor to the communications, but it is possible to render the communications totally unintelligible to all but the intended receiver(s).

There are many ways to reversibly transform data from its plain form to something that looks unintelligible, but many of these can be figured out (broken) by someone else. The study of how to hide secrets is cryptology. Trying to figure out the secrets that someone else has hidden is cryptanalysis. These two sciences are, of course, very much intertwined. History reveals many examples of cryptology that worked, and that didn't [KAH]. Successful cryptanalysis depends on taking advantage of as many of the following as are available to the cryptanalyst: (1) taking advantage of the redundancy in any natural language to determine the validity of assumptions, (2) clues gained from corresponding plain and cipher text, (3) information that might be known about the algorithm(s) used, (4) the general expected content of the cryptograms, (5) all of the cipher text that is available in the same system & key, (6) compromised keys, (7) as much computational and analytical power as can be obtained, and (8) mistakes made in the users of the cryptographic system. The cryptographer can make life as difficult as possible for the cryptanalyst by depriving him of some of these things by (1) using redundancy

reduction before encryption, (2) using an algorithm that is resistant to the known plain text attack, (3) & (4) using a strong enough algorithm that these clues aren't really useful, (5) changing keys often and selecting them properly, (6) guarding keys as closely as the data they protect justifies, (7) ensuring that there aren't enough computers in the world to do a brute force attack on the algorithm, and (8) making sure users of the system understand how to properly use it.

This thesis proposes one solution to the above challenge by proposing (1) an approach to linguistically based redundancy reduction, and (2) proposing a new data encryption algorithm (MPJ) that can be used where the National Bureau of Statndards Data Encryption Standard (DES) is in use now, but is more secure.

A. Motivation

There has been a great deal of discussion of the security of DES in the open literature. Most of it has been favorable to DES [MEY], but there are a few indications that it would be wise to supplement the aging DES and eventually replace it.

DES has been in use since 1977, and has been used in a large number of applications where people have had many possible motivations for trying to break this cipher. During this time, it is possible that someone has discovered a computationally feasible method for doing so [KRU]. Under such circumstances, it is highly unlikely that such a discovery would be made known. There is no way to really know this for sure, unless you are one of the ones who has broken DES. The closest thing that I have found to an open admission of breaking DES is a story of the FBI successfully decrypting a file of drug transaction records that were encrypted on a PC using a DES board [MCL]. The

DES board that the criminal used has an algorithm to generate a key from a word or phrase. By an exhaustive search of an English dictionary and key names from the criminal's family & friends using a supercomputer, the file was solved. This indicates some weakness in DES, but even more weakness in the way the key was chosen. The key was not chosen randomly, so the FBI's job was much easier.

DES is subject to attacks that require precomputation that could tie up a supercomputer for a few years, after which it would take only a few days to solve a DES cryptogram. This is becoming less of a barrier as the price of computers drops and the speed and storage capacity of computers increase.

The U. S. National Security Agency (NSA) is acting to release some of its own algorithms for ''sensitive but unclassified'' applications, such as communications between government contractors [NEW]. The NSA also releases some classified algorithms to chip manufacturers for use in classified devices, but under strict controls [ULT]. This will take some of the load off of the DES algorithm, but there is a catch. They intend to keep the algorithms secret and control the keys. Since the security of the algorithm is dependent on the key in a way that only the NSA knows, this gives the NSA the exclusive ability to read everybody's communications. That is not all bad, as it discourages the use of one of those systems by someone engaged in spying or other illegal activity. Unfortunately, these systems are not an option for protection of a corporations proprietary data that may have a great deal to do with profits and losses, but are not really in the domain of the NSA.

There are other alternatives to DES now, but none of them in the public domain are even as good, let alone better, for general cryptography. For example, RSA encryption (an exponential encryption algorithm named after the initials of its inventors) has great advantages in the authentication of digital signatures, but the complications of selecting good keys and the fact that the security of RSA relies heavily on the failure of the state of the art in mathematics to progress makes it at least inconvenient to use and at most insecure.

Because of the above considerations, it is my intention to suggest a better algorithm (MPJ) for general use in the private sector. Although the MPJ algorithm might be useful for government applications, too, the design requirements for the two areas of application vary [CHA] and the latter is kept shrouded in secrecy, so we shall leave it to the NSA.

B. Approach

First, the problem is defined. As explained in the above section, the basic problem is to create an algorithm to supplement DES. Design criteria are then conceived such that an algorithm that meets those criteria will solve the problem as defined. The design criteria chosen for the MPJ algorithm are discussed in chapter VI. To avoid repetition of one or more of the many mistakes that have been made throughout history with cryptography, it is, of course, necessary to research what was done in the past, both distant and recent. From this, many good ideas can be gleaned that apply to the problem at hand. These ideas, along with a knowledge of current technology are then applied to design criteria with a bit of creativity and a lot of hard work to define the new algorithm.

In the process of doing all of the above, it became apparent that the security of encrypted data was vastly improved by first removing as much redundancy from the data as possible. Therefore, as sort of a bonus, a linguistic approach to data compression is also presented that can either be used in conjunction with the MPJ encryption algorithm, another encryption method, or just by itself for the savings that it gains in communications channel capacity and/or data storage space.

II. HISTORY OF CRYPTOGRAPHY

One of the best works on the history of cryptography is *The Codebreakers*, by David Kahn [KAH]. In that book, David Kahn discusses cryptography from prehistory to World War II. The first codes and ciphers were in written form, and used to protect the privacy of communications sent by courier or mail through hostile or unknown territory. Some of these were reasonably good, but most were not difficult to break using manual methods, provided that the interceptor had sufficient cipher text and perhaps some probable text. The use of radio, especially by the military, increased the need for cryptography, as well as increasing the rewards for those who could break the encryption schemes in use. Kahn's documentation of the efforts of those who broke some very complex encryption schemes, like the German Enigma and the Japanese Purple Ciphers, lend great insight to the kind of process cryptanalysis really is.

Kahn points out the kinds of mistakes the inventors and users of cryptographic algorithms tend to make that reduce the security of their communications. For example, German users of Enigma tended to choose a three-letter indicator for their messages that consisted of three consecutive letters on the keyboard. This substantially reduced the number of keys that had to be searched to determine the one that they were using. While the designer of an algorithm may calculate the great number of combinations of keys that there are, the cryptanalyst looks at ways to isolate parts of the key so that the difficulty of a solution is much less than the size of the key space indicates. The difference in mind set between the concealer of secrets and the one who prys into them has caused many an inventor of an encryption algorithm to be overconfident.

The job of the cryptanalyst is a tedious one. He tries all kinds of things to try to unscramble the cipher text in front of him. Sometimes the search is fruitless. Sometimes the search yields something that looks like a meaningful language. It is this ability to recognize a meaningful message when it comes out of the various operations that the cryptanalyst tries that makes the whole process possible. It is also helpful for the cryptanalyst to know some probable plain text that is contained in a message. This is almost always the case. For example, military messages even now have a very stereotyped format, with the from, to, and date indicators in the same places in the message. The cryptanalyst almost always knows what language to expect a message to be written in, and this is a great help. Natural languages contain a great deal of redundancy. A message that is only 90% recovered is usually readable. Natural languages also have very consistent statistical properties that are very useful in cryptanalysis, especially when the cryptanalysis is automated. The only time that these things don't help the cryptanalyst is in the ''one-time pad.''

III. ELEMENTS OF ENCRYPTION

Several basic elements make up the basis of a multitude of possible encryption algorithms, either alone or in combination [BEK][BAL]. Although most of these elements may be used to form the basis of an encryption method all by itself, the most secure methods of encryption will use several of them. For example, substitution and permutation are basic elements of both the DES and MPJ encryption algorithms. These algorithms, in turn, are best used with some form of feedback.

A. Substitution

A substitution cipher simply substitutes one symbol from the plain text alphabet for another one from a cipher text alphabet. The plain text alphabet can be a natural language alphabet, ASCII, EBCDIC, Baudot, the set {0,1}, or any other finite set. Cipher text alphabets, likewise, may be any finite set. To be able to easily apply computer methods to the manipulation of these alphabets, it is preferable to use alphabets that are groups of binary digits (bits). This is not a severe limitation, since a correspondence can be set up between any finite set and a set of binary numbers.

1. Monoalphabetic

A monoalphabetic substitution cipher is one in which each letter of the plain text alphabet is always substituted for by the same cipher text alphabet. The cipher text alphabet need not be in any particular order. A subset of the monoalphabetic substitution cipher is the Caesar cipher. This cipher replaces each letter with a letter that is n letters later in the alphabet. This cipher is so named because it was reportedly first used by Caesar.

[KAH] With only 26 possible keys, this is obviously not very secure. In fact, any substitution cipher that substitutes one letter for another is vulnerable to solution by frequency analysis, and can be solved, given enough cipher text with the same key. How much cipher text is enough depends on the nature of the plain text, but for plain English, as many characters as there are letters in the alphabet is sufficient for the probabilities of occurrences of letters to betray enough letter identities to make solution probable. While ciphers of this type no longer have any value for serious cryptography, they do make fun puzzles for elementary school children. For example, the following cryptogram is presented for your solution:

UIJT JT B DBFTBS TVCTUJUVUO DJQIFS/ XPVME ZPV USVTU VPVS TFDSFST UP JU@

Substitution ciphers can be made more secure by operating on substitutions of more than one letter at a time. For example, the substitution could be made on digraphs or trigraphs (groups of two or three letters). A more practical example is the Data Encryption Standard. DES is actually a substitution cipher with an effective alphabet size of 2^{64}, or about 1.84×10^{19}, and the MPJ Encryption Algorithm is a substitution cipher with an effective alphabet size of 2^{128} or about 3.4×10^{38}. Not only is it difficult to get that large of a sample to analyze for statistics, but the memory required to store the substitution table is impractical even for computer systems using large optical storage disks.

2. Polyalphabetic

A polyalphabetic substitution cipher is one in which each letter of the plain text alphabet may be replaced a letter from one of many cipher alphabets. The selection of which alphabet to take the substitution from may be determined in many ways. The most usual method is by selecting the cipher alphabet by a combination of a message indicator and the number of characters from the beginning of the message. The classic example of this type of cipher is the rotor-based machine. The substitution occurs by physical wire crossings within rotors. After each character is enciphered, one or more rotors are rotated by a ratchet mechanism. The message indicator, which is part of the key, determines the starting position of the rotors. This effectively allows the use of more cipher alphabets than there are letters in the message. Because of this, the use of statistical analysis on any one message to guess at the substitutions is useless. There is, however a good way to attack this kind of cipher when it is used heavily. This is by analyzing the statistics of the first character in each message, then the second, etc. This is called analyzing the messages ‘‘in depth.’’

Although the rotor-based polyalphabetic ciphers used by Germany and Japan in World War II contributed greatly to their losses because their messages were read regularly by the allies, it is possible to create a more secure polyalphabetic substitution with computer technology. The primary weakness of the mechanical rotor machines was the infrequent changing of many of the parts of the key such as rotor wiring and ratchet construction. A more general method of using a different set of alphabets for each message would be very secure. Unfortunately, it would result in keys larger than the message. Since

the minimum key length for absolute, provable cryptographic security, as determined by C. E. Shannon [SHA], is exactly the length of the message, this solution is not very efficient with respect to key management.

B. Permutation

Permutation is taking the same symbols and writing them in a different order to confuse an interceptor. There are many ways of doing this geometrically. For example, words may be written vertically then transcribed horizontally, or they may be written on a strip of paper wrapped along the length of a rod of a given diameter, then unwrapped for delivery. One possible permutation of binary digits is shown graphically in figure III.B.

In general, a permutation operation may be considered as an operation on a block of symbols, where each symbol's position is changed to another position within the block according to some rule. The blocks may overlap each other if the permutation is undone from the end of the message to the beginning.

The rule(s) determining how the permutation is done may be fixed, or they may depend in some way on a key. Although historical permutation ciphers tended to be rather simple, computer methods allow them to be potentially rather complex, especially if the symbols that are scrambled are individual bits.

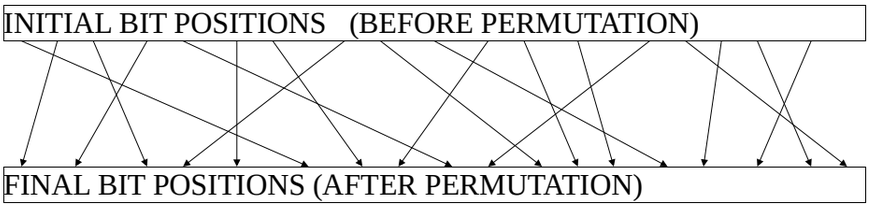


Figure III.B Permutation.

Permutation, when used in combination with substitution, is very useful in further increasing the complexity of a cipher, since the two algorithms act in different ways to baffle the interceptor. This is significant, since it is not possible to actually increase the security of a substitution cipher by simply repeating the same thing with a different key. The result in such a case would be another substitution cipher with the same complexity.

C. Noise Addition

Noise addition can either be a way of enhancing another encryption scheme, or of hiding the very existence of a message. See figure III.C for a block diagram of a noise addition process. For example, a grid can be set up on a piece of paper where only certain positions are significant. These positions are then concatenated together to reveal the secret message. The rest of the paper is then filled with something that uses those characters, but contains something different, like a letter to someone's mother. The same thing can be done electronically, by defining a block of bits with only part of the bits significant. The rest of the bits are then filled with truly random noise, like the output of a Geiger counter. This method can be very effective, especially if the position of the information-bearing bits vary from block to block in a pseudorandom manner, and if there is a relatively large proportion of truly random noise. The obvious disadvantage to this kind of scheme is that it makes very poor use of communications channels and data storage facilities.

Figure III.C. Noise Addition

One useful variation on the noise addition method is to multiplex several encrypted streams of data (encrypted with another method) together with some pseudorandom interleaving. Since the other encrypted streams of data act like the random noise added to any one stream of data, this is a good way to improve security when a large volume of encrypted data must be sent through the same channel.

D. Feedback & Chaining

A block cipher like DES or MPJ, when applied directly to an input file with a highly repetitive structure (i. e., lots of spaces between columns) will also display some of that structure. Although it may not be possible to determine the exact contents of what has been encrypted, it may be rather easy to determine something about the nature of what has been encrypted. To deny the interceptor even this information, and to further increase the complexity of the encrypted information, feedback and chaining may be used. Chaining refers to making the results of the encryption of one block dependent on previous results. This is usually done in one of two ways. One is by adding the plain text from the last block to the cipher text of this block, modulo two. The other is by adding the cipher text from the preceding block to the cipher text of the current block, modulo two. These are referred to as plain text feedback and cipher text feedback, respectively.

Figure III.D shows block diagrams of the basic modes that a block cipher can be run in. In this diagram, P refers to the plain text message, C the cipher text, and IV the initialization vector. The initialization vector is only used in the feedback modes to encrypt the first block of data. After the first block is encrypted, the feedback value is used instead.

1. Plain Text Feedback

Plain text feedback has the property that any errors in transmission get propagated clear through the rest of the message. This may be a desirable property if it is necessary to detect any tampering with the message, but it makes it difficult to use real (error-prone) channels. If it is necessary to have the property of error propagation to detect tampering and to use real channels, then the message should be transmitted using an error detection and correction protocol of some sort.

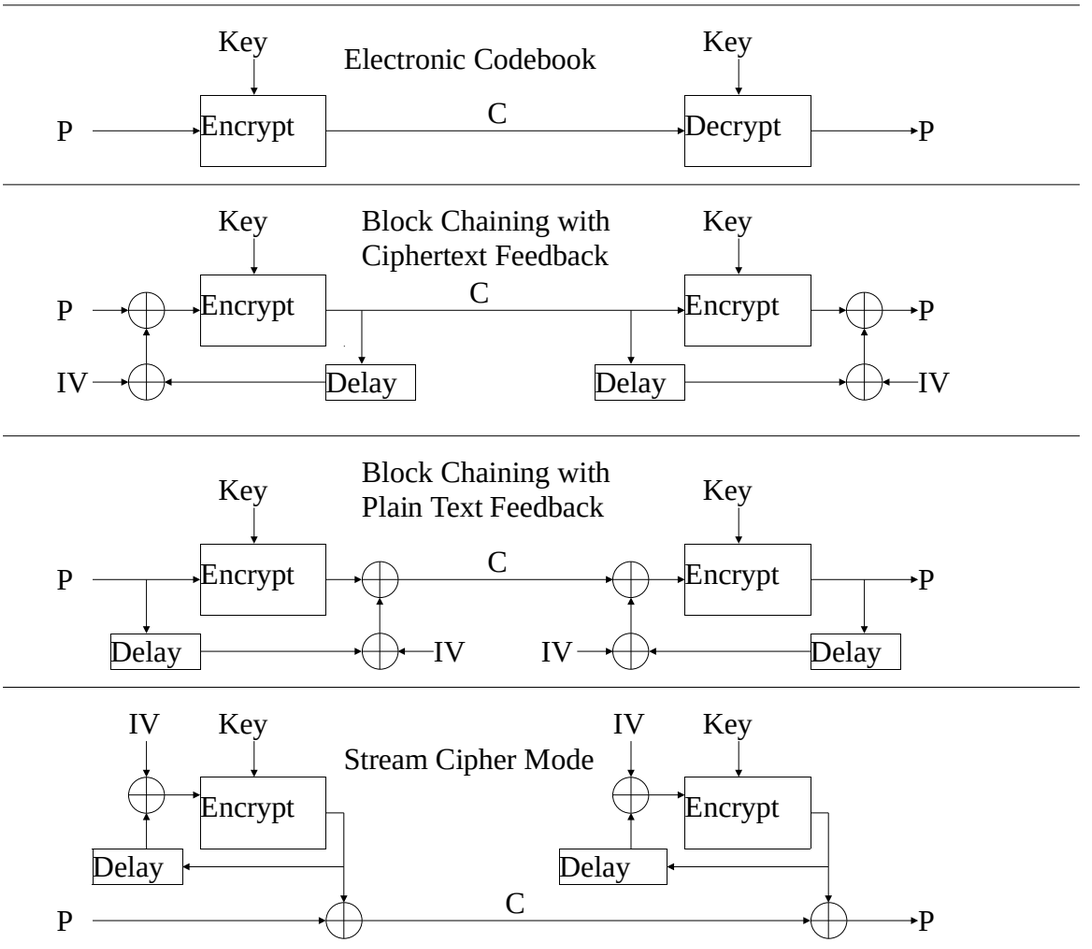


Figure III.D. Block Cipher Modes.

2. Cipher Text Feedback

Cipher text feedback also causes some error propagation, but not clear through to the end of a message. An error in one block will cause an error in that block in the bit positions where the error occurred and in all of the next block, but the receiving station can recover all subsequent error-free blocks. This method is a good compromise between security and error recovery. It is still desirable to wrap encrypted data in an error detection and correction protocol to avoid having a one bit error wipe out many bits.

E. Analog Encryption

There are several methods of analog encryption. None of them are as secure as digital methods can be. Analog encryption is generally based on spectrum inversion, spectrum scrambling, time slice scrambling, and (for video signals) suppression of synchronization information. These elements are also used in combination [LOD]. These methods are commonly used by cable TV companies (and sometimes on satellite channels) to ensure that people only get the programming that they have paid for. Articles on how to build devices to defeat some of these things appear periodically in electronic hobbyist magazines.

The state of the art in current use for protection of satellite TV signals is the General Instruments Video Cipher II. This device used digital (DES) encryption of the audio information and a partially analog encryption of the video signal. This device is a deterrent to those who receive satellite TV without paying a cable TV operator, but it has been broken by a hacker who figured out that it was possible to modify the microcode in the device so that a key purchased for one machine would work on all of them with modified

code. This is, of course, illegal, but it allows one person to pay for a service and many people to use it for free [DOH]. The problem here is not one of the encryption algorithm, but in the key management.

Figure III.E shows a general analog encryption scheme that rearranges both time and frequency blocks of a signal according to some pattern determined by a key.

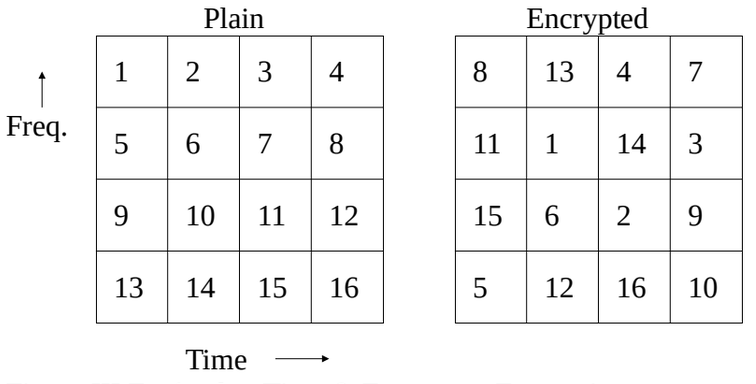


Figure III.E. Analog Time & Frequency Encryption

IV. FACTORS RELATED TO ENCRYPTION

A. Change of Language

Although it is not really encryption, the use of a different language from what the interceptor is likely to know does increase the difficulty of cryptanalysis. For example, I know that it would be far more difficult for me to solve even a simple Caesar substitution cipher in Chinese than in English, Spanish, or German. This principal was used effectively by the United States against Japan in World War II by using the Navajo language for spoken radio communications [KAH]. Since the Navajo language had no written form, and was only spoken by a small group of Native Americans, there were no Japanese who knew the language. The Japanese never did figure that ‘‘code’’ out. Thus, the Navajo ‘‘Code Talkers’’ made a great contribution to their country.

Part of the reason for the success of the Navajo Code Talkers was that the Japanese had no idea what was going on. Since this success is now well known, it is difficult to determine the potential success of a repeat of this kind of approach. There are still some languages that are spoken only by a very few people in the world, some of which have no written form. These could be used to increase the effective ‘‘key space.’’

Although a change of language was a great victory for the Navajo People and the United States of America during World War II, it is difficult to adapt this success to computerized communication methods.

B. Digitization

The very act of digitizing an audio or video signal provides some protection from the casual eavesdropper. Although this provides no protection against a determined snooper, it is an appropriate level of additional security for protecting the privacy of things like mobile telephone conversations. Of course, once an analog signal is digitized, there is a wide range of digital encryption methods available. Of course, some methods would be overkill, since the same communication would probably be transmitted in the clear over wire and/or microwave channels as well as by radio between the car and the stationary cellular transceiver.

Although there has been some ill-conceived efforts by the mobile telephone industry to legislate privacy for such applications by saying that it is illegal to listen to someone else's phone calls around 800 MHz (where cellular phones usually operate), this kind of thing is unenforceable. Worse yet, such legislation provides a false sense of security in a world where scanners and other receivers that operate in that range are readily available.

C. Compression

Data compression itself tends to obscure things by taking, for example, and ASCII text file in English and reducing it to a binary file that is not as easy to read. There is a more important effect of data compression, however. The removal of the natural redundancy of the plain text data before encrypting it with some kind of encryption algorithm makes it much more difficult to cryptanalyze the cipher text. In the extreme

case where all of the redundancy of a message is removed, all messages of a given length would be meaningful, and it would be impossible for the cryptanalyst to determine which one was intended.

D. Multiplexing

Multiplexed signals are less readable to the casual observer, but standard multiplexing offers no real increase in cryptographic strength. It can, however, increase the strength of multiple streams of encrypted data when the multiplexing is done with a pseudorandom interleave (as discussed earlier under noise addition). This makes the multiplexing and demultiplexing processes more complex with respect to synchronization and timing, and is likely to introduce more delay into the system than more straight forward multiplexing arrangements.

V. COMPARISON OF SELECTED ALGORITHMS

A. One-Time Key Tape

The one-time key tape, also called the one-time pad, has a tremendous advantage. It is the only commonly used encryption method that is provably secure. Proving the security of most other systems generally reduces to an impossible negative proof — an attempt to prove the lack of a method to defeat it.

The way the one-time key tape works is to add each character of the message to a character of the key modulo the alphabet size. When working with any binary stream of data, this is easy to implement by exclusive-oring the input stream with the key stream. At the receiving end, the same thing is done to decrypt the data. Since P + K + K = P (modulo 2), the original stream is recovered.

As implied by the name, it is important that each key be used only once. If it were used more than once, then the system would be vulnerable to attack with the known cipher text and corresponding plain text attack. The key is obtained from the boolean identity: K = (P + K) + P, where K is the key, P is the plain text, P + K is the cipher text, and all additions are modulo the alphabet size (usually 2). The key thus recovered would then be used to decipher the next message that used it.

According to C. E. Shannon [SHA], for a cipher to be provably secure, the number of keys must be as great as the number of potential messages. The number of keys he refers to, of course, is the number of keys that yields a distinctly different transformation of the message text into cipher text.

This method of encryption is provably secure, since an exhaustive search of the keys applied to any cipher text of a given length will yield all possible plain text messages of the same length. Since there are many messages of the same length that make sense, many of which have totally contradictory or unrelated meanings, the cryptanalyst has no way of knowing which one was intended unless he has the key.

Naturally, any method of encryption this secure has a price. The price is the sheer volume of keying material that must be kept secure and managed. For any communication network or data storage system that handles large amounts of data, key management becomes a nightmare with this system. Keys must be generated randomly — not pseudorandomly — to be absolutely secure. This means that a random process should be measured in generation of the keys.

A block diagram of the one-time key tape is shown in figure V.A.

Figure V.A. One-time key tape (AKA One-time pad)

B. Linear Shift Register Feedback

Linear shift registers with selected feedback taps are commonly used to generate pseudorandom sequences for such things as spread spectrum communications. They could be (and are, in some cases) applied to cryptography by exclusive-oring the pseudorandom stream with the plain text stream.

The main weakness to this approach is the cipher text with corresponding plain text attack. If the cryptanalyst obtains the plain text that came from a certain cipher text, then he can recreate a portion of the pseudorandom stream by exclusive-oring the two together. The linear shift register feedback function can then be expressed as a system of linear boolean equations that will solve the portion of the cycle that was received. A solution of this system of equations is possible with only enough bits to be a few times as long as the shift register at the most [DEN]. Since this amount of plain text required to break the cipher is so much less than the length of the pseudorandom sequence, it is likely that this solution can then be applied to additional cipher text to recover it, too.

C. Exponential Encryption

The classic algorithm of this type is the RSA algorithm, which is named after the initials of its creators (R. L. **R**ivest, A. **S**hamir, and L. **A**ldeman). The security of this algorithm rests on the fact that even with supercomputers, it is very difficult to factor the product of two very large prime numbers. The RSA algorithm has a unique property in that the key has two parts, one of which may be made public. This makes this algorithm very well suited to the use of digital signatures.

The RSA algorithm defines the relationships between the plain text message, the cipher text, and the elements of the key as follows [JAC]:

$$C = P^e \bmod m$$

$$P = C^d \bmod m$$

$$m = pq$$

where C = Cipher text, P = Plain text, e = encryption key, d = private decryption key, m = public modulus, and p and q are randomly chosen large (> 150 digits each) prime numbers. The receiver chooses a private key d and calculates a public key e. d must be relatively prime to (p – 1) and (q – 1). The algorithm works because ed = 1 mod the least common multiple of (p – 1)(q – 1). The algorithm is only secure if very large prime numbers are used. A number of 100 digits is too small — this size of number can be factored now with a multiprocessor technique developed by Arjen K. Lenstra of the University of Chicago and Mark Manasse of Digital Equipment Corporation [COS].

The key generation for RSA is a bit nasty, but is reasonable using Rabin's test for primality: Let $n = 2^r d+1$, where d is odd. Choose a at random from $1 < a < n – 1$. Accept n as prime if either $a^d = 1 \bmod n$ or $a^{2jd} = -1 \bmod n$ for some j such that $0 \leq j < r$, otherwise reject it.

RSA is excellent for public key and authentication use, but it does have certain disadvantages for general encryption use. The processing time required for key generation and for the encryption/decryption process makes it a poor choice for use with high data rates, although there are some reasonably efficient ways to do the exponentiation [CHI]. The complexity of the algorithm makes it more difficult to implement than many others [VAN]. The worst problem, though is the way that the complexity of solving the algorithm could be reduced by several orders of magnitude by mathematical research. For example, J. Sattler and C. P. Schnorr recently published some algorithms that would appear to reduce

the complexity of solving an RSA key by a factor of about 10^4 [SAT]. Granted that we are talking about taking 9×10^8 years instead of 1.5×10^{13} years, but it is the potential for other discoveries that may drastically reduce this time even further that is the real threat.

D. Knapsack

A knapsack or trapdoor algorithm is one that is relatively easy to perform, but which is very difficult to invert. The RSA algorithm is one such algorithm. Another one, proposed by Ralph C. Merkle, uses the following vector operation: given an integer s and an integer vector $\mathbf{a} = a_1, a_2, ...,$ an find a vector $\mathbf{k} = x_1, x_2, ..., x_n$ where x_i is in $\{0,1\}$ such that $s = \mathbf{x} * \bar{a}$. (* denotes dot product.) [MER] This algorithm is referred to in another paper as having been broken [VAN]. Another algorithm using matrix operations is suggested by H. Retkin [RET]. This algorithm may be good, but it is not obvious to me that no one will come up with a good solution to breaking it, too. Therefore, I prefer to use encryption algorithms that have simple brute force solutions that are well understood, but just take too long to perform to be of concern.

E. Rotor Machines

Rotor machines were once the state of the art in cryptography. They are a way to form a polyalphabetic solution automatically. An excellent history of these machines is contained in [KAH], [DEA], and [KOZ]. Although they are obsolete now due to the superiority of many computer algorithms, they played a major role in the history of cryptography.

There were many different variations on the rotor machine, but the main principle of each of them was that the input, which is normally indicated by one of 27 (or whatever the alphabet size in use is) wires is connected to a voltage level by a keyboard. This wire is then connected via rotary contacts to a rotor that physically permutes the position of the wire. This causes a simple substitution for each character, or a poly alphabetic substitution. The output is taken from rotary contacts on the other side of the rotor. Several of these rotors are placed in series. After each character is encrypted, one or more of the rotors is moved to a new position by a ratchet mechanism. The net effect is a different simple substitution for each letter in the message. A typical rotor-based rotor machine is shown in figure V.A.1. Typical rotors with variable wiring between contacts are shown in figure V.E.2. Common additions to this scheme are a plugboard style permutation performed

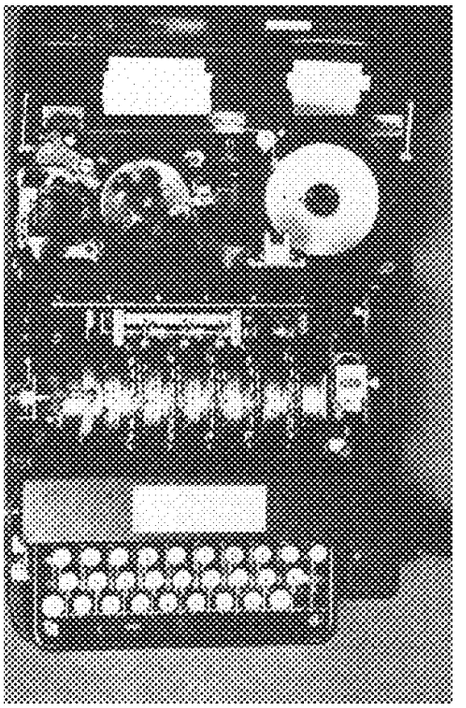
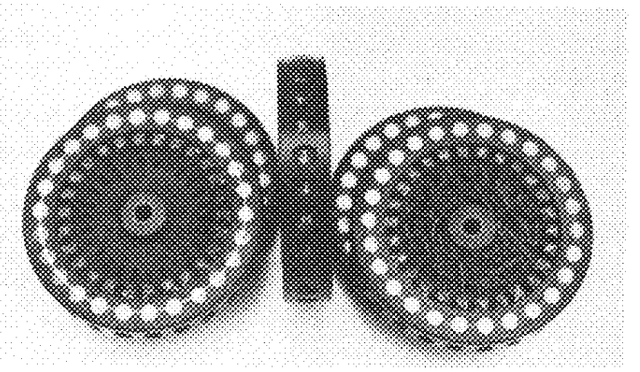


Figure V.A.1 Typex Rotor Machine

Figure V.E.2. Wired Rotors



before or after the rotors and the addition of a reflecting rotor to run the signal back through the rotors in the opposite direction, using each rotor twice.

The key used is a combination of several things: (1) the permutation performed by each rotor (changed by rewiring it), (2) the order in which the rotors were assembled, (3) the starting position of the rotors, (4) the plugboard connections, (5) the number of rotors used, and (6) the ratchet arrangement. Taken all together, these form a large key space. If the entire key space is varied simultaneously and for every message, then this polyalphabetic substitution is very secure. Due to the construction of these machines, some of the variables were not as easy to change as others. Therefore in practical use, the variables were not changed all at once. For example, the number of rotors and the ratchet arrangement would probably be fixed for the life of the machine. Therefore, if this portion of the key is solved for once, that is enough until a new kind of machine replaces it, perhaps years later. The rotor wiring is tedious to change, and therefore unlikely to be changed very often. The order of the rotors might be changed periodically, but during heavy use, the only thing that was changed for each message is usually the starting position of the rotors.

Using an alphabet size of 26 and five rotors, varying only the starting position of the rotors provides a key space of less than 12 million, which is within the range of possibility of solution by mechanical computer, and a quick task for one of today's PCs. With many messages encrypted with the same rotors, the rotor wiring can be solved by frequency analysis ''in depth.'' In other words, the permutation applied to the first letter of each message can be solved by determining the frequency of occurrence of each code

letter and comparing that with the language of the source language. This can be done for each position in the message. From these substitutions, the rotor wiring and plugboard connections may be deduced. This is a more lengthy process than solving for the rotor starting position, but once done the solution is likely to be good for a while. Using this kind of approach, isolating the parts of the key, the Allies read many German and Japanese messages during World War II.

Rotor machines can be simulated in a computer program with more flexibility than in mechanical hardware, with many improvements. There are, however, better methods to use in computer algorithms that were not easy to use in mechanical devices. Therefore, rotor machines are interesting to study for historical value, but they are obsolete at a time when computers are becoming almost as common as televisions.

F. Codes

Codes, as opposed to ciphers, operate on linguistic units like words, phrases, and sentences, rather than directly on the units of the alphabet. Codes have the advantage that they generally compress the message. For example, the codeword ‘‘APPLE’’ might mean ‘‘The supplies will be shipped on Monday via private courier’’ and ‘‘GRAPE’’ might mean ‘‘The bid is too high. Reduce it by 10%.’’ If there are only a few different messages that might be encoded, a substantial savings in space can be realized.

Codes are useful for compression of information, even when no encryption is intended. For example, the Uniform Commercial Code, used for telegraphy saves on tolls, even though it is published and gives no real security. Another example is the use of Q signals on amateur radio, where encryption is illegal but brevity is important, especially

when using a slower mode of communication like Morse Code. QST means ‘‘The following is a message of interest to all Amateurs’’ and QSL means ‘‘My location is _____.’’

When used to hide the meaning of the message, the code must be changed often. This is much more difficult to do than to change the key to an encryption algorithm. Therefore, this technique has limited value except for some diplomatic and military codes. For commercial applications where brevity and security are both required, it is easier to first encode the plain text with a fixed code to reduce its size, then to encrypt the encoded text with an encryption algorithm whose key can be easily changed. This technique is more secure than the use of the encryption algorithm alone, even if the code used is widely known. See chapter VIII for more on the effects of data compression when used with encryption.

G. Galois Field and Hill Cryptosystems

A Galois Field cryptosystem is one in which the letters of the encryption alphabet are assigned arbitrarily to the polynomials of degree n. These polynomials are then operated on in blocks of m letters by matrix multiplication with a matrix that represents monic irreducible polynomials of order n+1. All operations are done modulo p. To decrypt the data, the ciphertext is multiplied in the same manner by the inverse of the encryption matrix (obtained using the same modulo arithmetic). The resulting numbers are then substituted back for the letters they represent.

The Hill cryptosystem is similar, except that n is fixed at one, and that p need not be absolutely prime as long as several integers less than p are relatively prime to it. For more details on how these cryptosystems work, see the articles by Hill, Cooper, and Patti [HIL] [HLL] [COO] [PAT].

Both of these systems are based on modular linear algebra. Therefore, they are subject to solution of the key in use by linear algebra if enough cipher text and corresponding plain text is available. If the permutation of the encryption alphabet in use is known, then the amount of corresponding plain and cipher text needed for a solution is not much greater than the key length. If the permutation of the encryption alphabet in use is not known then there must be a sufficient quantity to also perform statistical analysis attacks on the alphabet in use. Therefore, I do not recommend the use of these systems for serious encryption because of these vulnerabilities. Not only are they less secure than their key length (the combined length of the coefficients of their matrices is less than the effective key length of $\log_2(m2^m\Sigma^m_{i=1}p^i)$) would indicate, they are computationally less efficient than DES and MPJ. Key generation for these systems becomes increasingly complex as the size of the key grows, too, because the probability of coming up with a valid set of monic irreducible polynomials that form an invertible matrix decreases rapidly with matrix size.

These cryptosystems, do, however, provide interesting mathematical illustrations on what can be done with Galois Fields. They also provide adequate security for cases where the information being secured is not of great commercial value, and is therefore unlikely to be cryptanalyzed.

H. DES

The National Bureau of Standards (now known as NITS) Data Encryption Standard (DES) was published in the Federal Information Processing Standards Publication Number 46, dated January 15, 1977 (FIPS PUB 46) [DES]. For details, please refer to that publication. A summary of the algorithm follows.

Encryption consists of an initial permutation, sixteen rounds of encryption, then an inverse of the initial permutation. See figure V.H.1. Each of the sixteen rounds of encryption consist of taking the right half of the input block (32 of the 64 bits) and running it through a nonlinear function of the 32 bits and an internal key, then adding this result to the left half of the input modulo two. This 32 bit answer becomes the next round's right half block. The next round's left half becomes the right half block without modification. The nonlinear function used consists of a bit selection E that selects 48 bits from the input of 32 (several of the bits are repeated). These 48 bits are added modulo 2 to the round key of 48 bits. See figure V.H.2. The result of that operation are then fed six bits each into eight substitution boxes. Each of the eight substitution boxes are different, but the same set of eight boxes

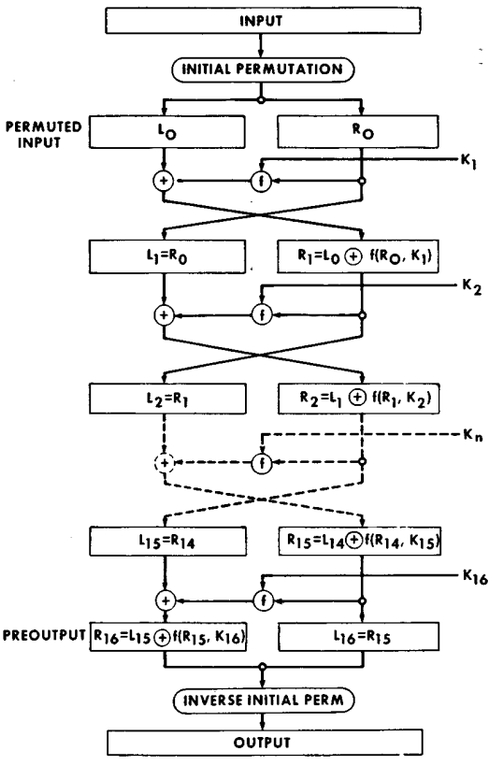


Figure V.H.1. DES Enciphering

are used for each round. Each substitution box gives an output of 4 bits. The output of these boxes are fed into a permutation P that rearranges the output in a fixed manner.

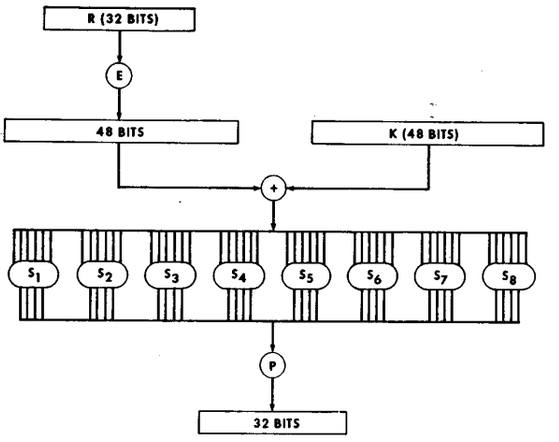


Figure V.H.2. DES Nonlinear Function

The sixteen internal keys are generated from the 56 bit input key by feeding the input key into a fixed permutation that rearranges the order of the key bits. The key is then split into left and right halves called C and D. Each half is shifted left one or two times (according to a fixed table) before generating each internal key. Each of the sixteen internal keys is generated by taking the two halves of the key as shifted and permuting them in a fixed manner. See figure V.H.3.

The key and the resulting internal keys are the only things that vary in this algorithm. The initial and final permutations and the contents of each of the substitution boxes are constant. The two permutations used in generating the internal keys are constant. The bit selection and permutation used within the nonlinear function are constant.

The strengths of the DES is that its cryptographic strength depends only on the key, that the algorithm is easy to implement in a single IC, that it has been well tested an no one has publicly announced a solution, that hardware and software that uses it is readily available, and that the algorithm places very few restrictions on key generation so that random numbers may be generated by the users for use as keys.

The weaknesses of the DES are that the key is too short for security in the face of anticipated increases of computing power, that it is old enough that it is likely that someone has broken it (found a short-cut solution), that hardware implementations of the DES are too slow for some applications, and that it limits itself to be simpler than is really necessary with current technology.

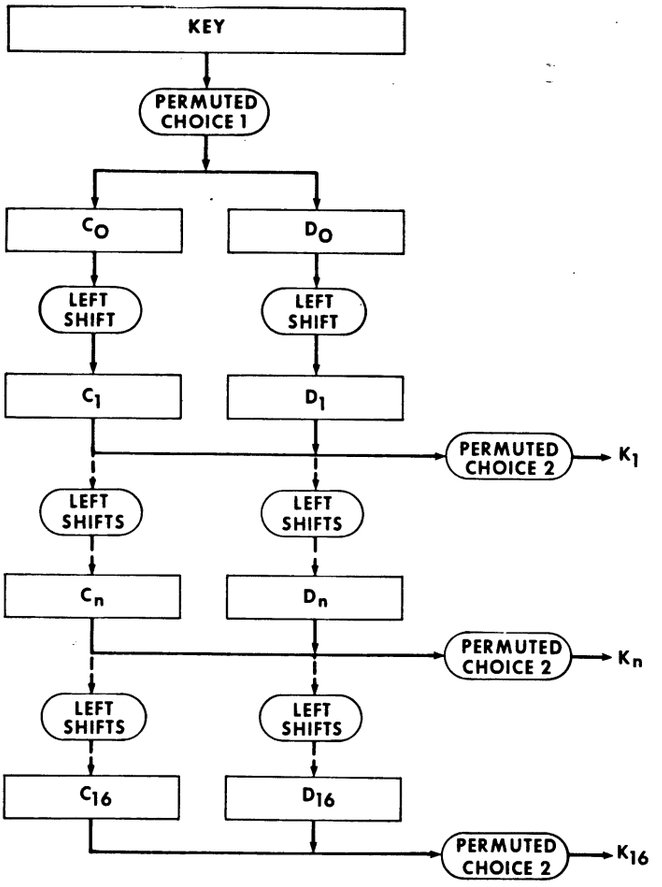


Figure V.H.3. DES Internal Key Generation

VI. DESIGN CONSIDERATIONS FOR MPJ

The following paragraphs describe the design considerations that I used in creating the MPJ Encryption Algorithm, and the reasons for each.

A. Strength Based on Key

The strength of the system must rely on the security of the key only. It cannot depend on the algorithm being kept secret, because the algorithm will be published. Even if the algorithm were not published, it would probably be reverse engineered from software implementations of the algorithm. The algorithm must be constructed in such a way that there is no computationally feasible way to derive the key from samples of corresponding plain text and cipher text.

B. Usability of Random Keys

The key selection should be as easy as the random selection of a number in a given range. Selecting a very secure key should be no more difficult than flipping a coin once for each bit of the key, or generating keys using a pseudorandom sequence combined with random events such as timing of keystrokes on a computer. A one bit change in the key should provide a drastically different transformation, so that a potential cryptanalyst has no idea when a key that he guesses might be ‘‘close’’ to the right one.

C. Key Length & Block Size

The key length should be significantly longer than the DES 56 bit size. A key size of 128 bits (sixteen 8-bit bytes) was chosen as being very manageable, yet highly secure even when attacked by multiple array supercomputers. The block size was also chosen as 128 bits (twice the size of DES) to provide a significant increase in complexity of the encryption.

D. Effort Required to Break

The effort required to break the algorithm by any method should be so great as to make such a task unfeasible even if significant advances are made in computer technology.

This requirement is intimately linked to the choice for key size and block size. For example, if one thousand keys could be tried every nanosecond (10^{12} attempts per second), an exhaustive key search that resulted in success after only testing one millionth of the possible keys (extremely good luck) would take $2^{128}/((10^{12})(10^{6})) = 3.4 \times 10^{20}$ seconds = 1×10^{13} years. If someone figured out a computational method or weakness in the algorithm that reduced the complexity by a factor of a million, and that someone also figured out how to compute the solution a thousand times faster, and they still believed in one in a million chances at good luck, they could possibly come up with a solution in only 10,000 years. By way of contrast, testing keys at the rate of 10^{12} per second for DES with its 56 bit key, all of the keys can be tested in 7.2×10^{4} seconds, or about 20 hours. While trying a trillion keys a second is not realistic with current general purpose computers,

current technology could be applied to dedicated, highly parallel encryption/decryption engines that could approach that speed — at a cost that would be prohibitive for all but large governments at today's prices.

Other than time, the other main measure of complexity of an algorithm is the storage capacity required to implement a lookup table attack. Suppose a lookup table were to be constructed that contained the encrypted version of just one block of cipher text corresponding to a very common block of plain text (such as 16 ASCII spaces) for every key possible. For the MPJ algorithm, this would take 2^{128} x 128 = 4.356 x 10^{40} bits. If this memory were constructed with some kind of device that could store one bit per atom of silicon, this would take 4.356 x 10^{40} bits x 28.086 amu/atom x 1.660531 x 10^{-24} grams/amu x 10^{-6} metric tons/gram = 2.03 x 10^{12} metric tons of silicon. That is about one thousandth of the mass of Deimos, the smaller of Mars' two moons [CRC]. For DES, the same table would only require about 215 micrograms of silicon. Nobody has come up with memory that dense, and fundamental physical limits make such a task difficult, indeed. It is not difficult to conceive of some breakthroughs in optical storage technology coming close, say using a thousand molecules of something per bit. Under those circumstances, DES would be vulnerable to such an attack, where MPJ would still be safe.

E. Computational Efficiency

The MPJ encryption algorithm must be computationally efficient enough to be implemented in software on a standard IBM PC or compatible (or on an Apple computer of comparable power), and fast enough to handle at least 10 megabits per second when

implemented in dedicated hardware. Note that this is less restrictive with respect to the hardware for DES, which was required to be simple enough to implement on a single chip using 1970s technology.

F. Communication Channel Efficiency

The MPJ encryption algorithm must not significantly increase the size of the plain text when encrypting it. This precludes the use of noise addition as a technique to be used.

G. No Back Doors or Spare Keys

While it may be impossible to guarantee that no ‘‘back doors’’ or ways to decipher a message without the key exist, the algorithm should be a sufficiently complex combination of simple, well-understood operations that no help is offered to the cryptanalyst from the structure of the algorithm. Spare keys (the situation where more than one unique key will decipher a message) are avoided by making the number of keys possible much less than the number of possible transformations that can be done on a set of blocks. This is true for MPJ because $2^{128} << 2^{128}$**!**.

VII. MPJ ENCRYPTION ALGORITHM

A. Description

The MPJ Encryption Algorithm can be looked at from the outside like a rather large electronic codebook that operates on 128 bit blocks. The algorithm may be used in several modes. It can be used directly in electronic codebook mode, in block chaining with ciphertext feedback mode, in block chaining with plain text feedback mode, and in stream cipher generation mode.

Given a 128 bit key, instructions to encrypt or decrypt, and a 128 bit input block, a unique 128 bit output block comes out. Encryption and decryption are inverse operations that can be done in either order. For example, if P is the plain text block, C is the cipher text block, E_{K1} is encryption with key 1, and D_{K1} is decryption with key 1, then $C = E_{K1}(P)$; $P = D_{K1}(P)$. A different cipher text results if the plain text is decrypted first, but $D_{K1}(E_{K1}(P)) = P = E_{K1}(D_{K1}(P))$. Multiple encryption can be done with several keys, as in these examples with three:

$C = E_{K1}(E_{K2}(E_{K3}(P)))$; $P = D_{K1}(D_{K2}(D_{K3}(C)))$ or a different method:

$C_1 = E_{K1}(D_{K2}(E_{K3}(P)))$; $P = D_{K1}(E_{K2}(D_{K3}(C)))$

1. Overall Structure of MPJ

Before encryption or decryption occurs, the substitution boxes are filled based on the input key.

To encrypt a 128 bit input block, it is run sequentially through ten rounds of substitution. Each round of substitution operates on the 16 eight-bit bytes that make up the input block individually. In between each round of substitution is a round of

permutation (wire crossings), for a total of 9 rounds of permutation. See figure VII.A.1. The permutation operation is identical each time. Each operation (substitution and permutation) has an inverse operation. Decryption is done by running the inverse operation of each of the above steps in the opposite order.

After one round of permutation, each bit in the output block depends on every bit in the byte that it is in and on eight bits of the key. After a round of permutation and the second round of substitution, each bit in the block depends on eight bytes of the input block and on eight bytes in the key. After the third round of substitution, each bit in the block is functionally dependent on 15 of the input bytes and on 15 bytes of the key. After the fourth round of substitution, each bit in the block is functionally dependent on every bit of the input block and on each bit of the key. After the tenth round of substitution the functional dependence on every bit of the key and the input block is so complex that it would be infeasable to determine the key used, even if the cryptanalyst has known corresponding plain and cipher text and knows the algorithm used. Because the substitution boxes are totally nonlinear functions, the problem of finding each of the 40,960 internal key bits is a tough one, indeed. It would be easier to find the original key of only 128 bits by brute force — something that is difficult, indeed.

2. Substitution Boxes

The substitution boxes are designed to be big enough to provide a large number of possible arrangements, but small enough to still fit comfortably within the memory of an MS-DOS computer. Each substitution box is therefore a collection of 16 substitution

boxes that operate on only 8 bits at a time, with a total of 256 entries. The substitution boxes may be an array or look-up table in a computer, or they may be implemented in hardware using RAM.

The substitution boxes are filled during the internal key generation process with a permutation of numbers that depends on the main key in a different way for each substitution box.

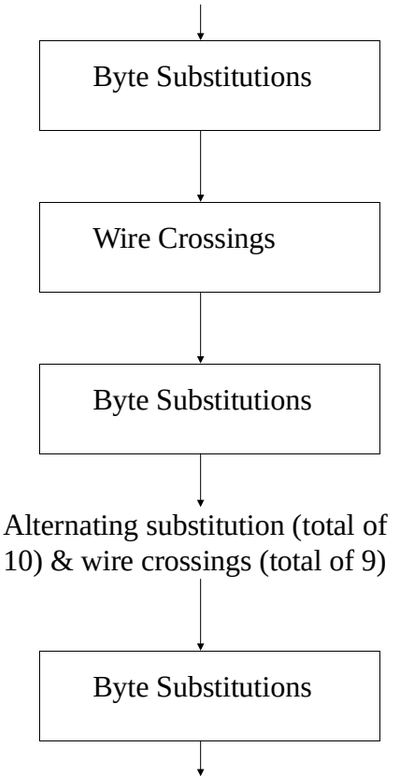


Figure VII.A.1. Overall Structure of MPJ

3. Wire Crossings

The wire crossings serve to extend the functional dependence of the output block on the input block across the internal byte boundaries. This is done by making each byte of the output of the permutation to contain one bit from each of eight different bytes. The selection of bits was chosen to make a computer implementation fairly efficient. A pure hardware implementation of this operation is, of course, much faster, since it involves only the propagation delay of the signal from one end of a short wire to the other. The input block bit positions are numbered from left to right (MSB to LSB) as 127 down to 0, then the new positions those bits occupy after the wire crossing are (MSB to LSB):

55, 46, 37, 28, 19, 10, 1, 120, 111, 102, 93, 84, 75, 66, 57, 48, 39, 30, 21, 12, 3, 122, 113, 104, 95, 86, 77, 68, 59, 50, 41, 32, 23, 14, 5, 124, 115, 106, 97, 88, 79, 70, 61, 52, 43, 34, 25, 16, 7, 126, 117, 108, 99, 90, 81, 72, 63, 54, 45, 36, 27, 18, 9, 0

The permutation is perhaps easier to understand graphically. In figure VII.A.3, the source bytes for the bits in 5 of the destination bytes are shown. The least significant byte of the output gets its least significant bit from the least significant bit of the same byte of the input. The next most significant bit comes from the corresponding bit of the next input byte to the left, and so on, until all eight bits are filled. The least significant byte is considered to be to the left of the most significant byte.

4. Key Generation

Internal key generation in the MPJ encryption algorithm consists of filling all of the substitution boxes (arrays in software or RAM in hardware). There are 16 substitution boxes used for each of 10 rounds. Each substitution box contains 256 entries of 8 bits each. Therefore, the actual internal keys have a combined length of 16 x 10 x 256 x 8 = 327,680 bits. These are obtained by manipulation of the 128 input key bits. Note that trying to attack the MPJ encryption algorithm by brute force trial and error using internal keys instead of using the 128 input key bits is ridiculous. The calculations given above indicate how difficult even a 128 bit random key is to solve by brute force. The only way that an attack on internal keys is useful is if there are parts of the internal keys that can be solved for separately, without having to solve for as many as 128 bits at once.

Attacks against the internal key structure are made computationally intractable in three ways. First, the smallest chunk of the internal key that could be meaningfully solved for is the contents of one of the substitution boxes. Each of these contain 256 bytes, or 16 times the length of the input key. Remember that the addition of each bit doubles the complexity of the solution, so this is not attractive compared to solving for the input key. The second impediment to this attack is to make the relationship between the contents of the substitution boxes and the input key quite nonlinear. The way this is done is by using a combination of rotating bit selection and the use of the last substitution box filled to compute the contents of the next one. The third line of defense against the solution of internal keys is the use of ten nonlinear rounds of encryption to make any attempt at constructing a system of linear equations to solve for the internal key given known plain text and corresponding cipher text infeasable. Note that each round of encryption in MPJ causes the whole 128 bit block to be changed, where each of the rounds in DES only modifies half of its 64 bit block. Therefore the internal structure of MPJ is very difficult to solve for, with each bit of the output having a nonlinear functional dependence on every bit of the input and on the contents of 1 + 8 + 15 + (7 x 16) = 136 of the substitution boxes,

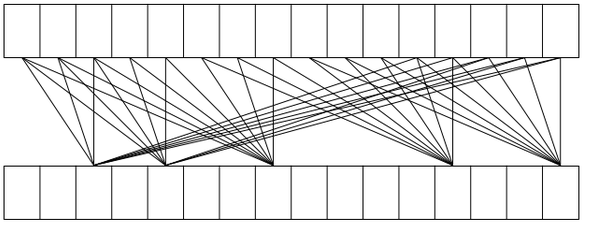


Figure VII.A.3. Wire Crossings
Source bytes for 5 destination bytes are shown.

containing a total of 34,816 bits. If the substitution boxes were linear, this solution would be possible, but there is no method I know of to solve such a system of nonlinear equations, either here or the simpler case of the DES internal structure.

To find the inverse of the substitution box functions, separate inverse substitution boxes are formed that place each address of a substitution box at the address of the data contained in the substitution box. For example, if the contents of the first substitution box in the first round of encryption for the input value of 1 is 219, then the contents of the first inverse substitution box in the last round of decryption for the input of 219 will be 1. In equations, SI[i, j, S[i, j, k]] = k for all i, j, k, where the first index of the array is the round of encryption, numbered from 0 to 9, or the round of decryption, numbered from 9 to 0 (defined differently for programming convenience); the second index is the position of the 8 bits that the substitution applies to in the input block, numbered from 0 to 15, left to right; and the third index is the value of the 8 bits input to the block.

The actual algorithm for substitution box filling consists of three main parts. The first one is a nested loop to fill one substitution box, permute the key with the same permutation used for the encryption operation, then to run each byte of the key through the substitution box that was just filled. This is done for each round of encryption, from 0 to 9, with each substitution box used in one round of encryption, from left to right, done within each of these rounds. In other words, the index of the three dimensional array that selects the position of the substitution box within each array varies faster than the index that selects the number of the round of encryption that the box uses.

The second algorithm is the one that fills each substitution box based on the contents of the input key, as modified by the previous steps. This algorithm uses bit selection and rotation to determine where each value of the output of the substitution box goes. To be an invertable function, each of the possible output values from 0 to 255 must appear exactly once somewhere within the array with 256 possible slots, numbered from 0 to 255. The first number placed in the array is 255, and there are 256 possible places to put it. The second one is 254, with 255 places to put it. This is continued until the last value, 0 is placed in the one remaining slot. For the first half of the values, the formula pos = (n * m) div 255 is used to determine which of the unfilled slots (counting in index order from 0 to n) is to be used to contain n. The variable pos is the position (counting only unfilled slots), * denotes multiplication, and div is the integer division operator (remainders are ignored). The variable m is determined for the first value of n (255) by selecting one bit from each byte of the key, starting with the least significant bit from the least significant byte of the key. The next bit (place value of 2) is selected from the next most significant bit of the key, and so on until m has eight bits. For the next value of n (254), the same thing is done, except that the bit selected is one bit position to the left of the one selected last time. The place value of the bit selected is the same in m as it is in the byte it came from. The bit to the left of the MSB in a byte is the LSB of the byte to the left of it. The byte to the left of the most significant byte is the least significant byte. For the values of n from 127 to 0, then the same thing is done, except that only seven bits

are selected for m, and the position is determined from pos = (n * m) div 127. For a more concise explanation of this process, see the commented pascal procedure *makesbox* in the program in appendix A.

The third algorithm used in internal key generation is the generation of inverse substitution boxes. This is done with a simple nested loop that fills the inverse substitution boxes according to the formula si[i, j, s[i, j, k]] = k for i from 0 to 9, j from 0 to 15, and k from 0 to 255. The array *si* is the collection of inverse substitution boxes, and the array *s* is the collection of substitution boxes filled by the above two algorithms. Filling of inverse substitution boxes is only needed for decryption mode.

B. Implementation in Pascal

Pascal source code for a program to implement the MPJ Encryption Algorithm is given in Appendix A.

1. Exceptions from Standard Pascal

This program was written for MS-DOS computers and compiled using Borland Turbo Pascal 5.0. To adapt this program for use on another system, note that the following features of Turbo Pascal that do not conform to the ANSI/IEEE770X3.97-1983 Standard Pascal were used in this program:

(i) The *assign* procedure is used to associate an operating system file name with a Pascal file name.

(ii) The nonstandard file handling procedures *blockread, blockwrite, close,* and *seek,* were used. The nonstandard function *filepos* was used.

(iii) The type *longint* (a 32 bit integer) was used for input handling to allow range checking without invoking a system error for a number that was only slightly out of range. A 16 bit integer could be used instead.

(iv) A generic file type was used, which is not defined in ANSI Pascal.

(v) The nonstandard operators *shl* and *xor* were used. The *shl* could be replaced with integer division by 2 (div 2), and the *xor* could be replaced by expressions of the form ((A *and* (*not* B)) or ((*not* A) *and* B)).

(vi) Logical operators (*and*, *or*, *not*, *xor*) were used with integers to perform bit-wise operations.

(vii) The + operator was used to concatenate strings.

(viii) The *uses* clause links in separately compiled units.

(ix) The include comment *{$I filename}* includes additional source code to be compiled with the current file.

(v) Additional nonstandard features were used in writing *startup* and the file handling functions that are unique to MS-DOS. These functions would best be rewritten for a different target system.

2. Main Program

The main program calls *startup* to initialize variables, get the key from the user, and determine which files are to be encrypted or decrypted. It then calls *makesbox* to generate the internal keys (fill the substitution boxes). If the decryption mode is to be used, it calls *makesi* to fill the inverse substitution boxes. It then does the actual encryption or decryption.

The main program allows for the encryption of multiple files (specified using MS-DOS wildcards), and passes these files to the encryption routine one block at a time. To provide additional security, files are encrypted in place, with each block of output overwriting the corresponding input block in the same place on the disk. If it is desired to keep a copy of both encrypted and plain text versions of the file, the input file should be copied before running this program.

Because this program uses block chaining with ciphertext feedback, only the encryption mode of the MPJ algorithm is used, the source of the feedback (input or output) being determined by the *decryption* boolean variable. This mode is used because (1) it obscures repetitive patterns in the source file that the electronic codebook mode might not hide, (2) it accommodates files of any number of bytes (including a short block at the end) with no complications and results in an output file that is the same length as the input file, and (3) it is slightly faster in operation because the procedure *makesi* does not have to be called. This mode does require an initialization vector, but since the initialization vector is encrypted before use, a constant initialization vector may be used.

To use the electronic codebook mode, delete the *for* statement following the calls to *encrypt*, replace one call to *encrypt* with a call to *decrypt*, and remove the comment delimiters from around the call to *makesi*.

3. Procedures Encrypt & Decrypt

These procedures directly reflect the overall structure of MPJ. They are simply calls to the routines that perform the substitutions and permutations, calling them in the proper order and passing a parameter to the substitution routines that select the right set of

substitution (or inverse substitution) boxes for the operation being done. The procedure *encrypt* makes 10 calls to *substitute*, with 9 calls to *permute* (one call to *permute* between each pair of calls to *substitute*. The procedure *decrypt* makes 10 calls to *isubst*, with 9 calls to *ipermute*, done in the inverse order of encryption.

4. Procedures Permute & Ipermute

Permute just scrambles the order of the bits in the 128 bit block in such a way as to maximize the dependence of each byte on every other byte of the input. Each byte is formed by selecting the least significant bit from the least significant bit of the corresponding input byte, then the next most significant bit from the next byte to the left, and so on. The least significant byte is considered to be to the left of the most significant byte. *Ipermute* is just the inverse of this permutation, with bytes being selected to the right instead of the left.

5. Procedures Substitute & Isubst

These procedures are simple array lookups in which each byte of the input block is used as the third index of the array *s* for the procedure *substitute* or *si* for the procedure *isubst*. The output is the contents of the array. The first index is the round of encryption or decryption (counting backwards if it is decryption), and the second index of the array is the byte position (0 to 15) within the input and output blocks.

C. Implementation in Hardware

For use with high data rates (greater than 10 Megabits per second), it is desirable to implement the MPJ algorithm directly in hardware. This also provides the advantage of greater security against tampering than a program on a general purpose computer might be subject to.

The hardware implementation consists of six basic elements: (1) Key generation, (2) serial to parallel conversion, (3) substitution, (4) wire crossings, (5) parallel to serial conversion, and (6) timing control. The key generation is done by a program running in a microprocessor. This program takes the key as input and fills the substitution boxes, which are simply static RAM chips. Serial to parallel conversion and parallel to serial conversion is done with shift registers. The wire crossings are done with physical wire crossings (or printed circuit board trace crossings) — a technique that is hard to beat for speed. Timing control is done with a sequential circuit that is synchronized to the incoming data stream.

The output data stream will, of necessity, be delayed by at least the block size of 128 bits, and some kind of protocol must be used to synchronize the blocks at the sending and receiving ends.

The figure VII.C shows a block diagram of a hardware implementation of the MPJ Encryption Algorithm. There are, of course, many variations possible. For example, block chaining could be used with hardware exclusive-or gates to do the modulo 2 addition and flip-flops for unit delays.

D. Strengths & Weaknesses

The main strengths of the MPJ encryption algorithm are:

- It is much more secure than DES.
- When used in conjunction with data compression, MPJ is probably more secure than many of the best military and diplomatic codes & ciphers.
- It is easy to generate a key for MPJ.
- It is capable of high data rates when implemented in hardware.
- It will run on a personal computer.
- It takes more time to generate internal keys from an external key than to actually encrypt or decrypt data, making exhaustive key search attack more difficult.
- The algorithm is published and may be freely used for any legal purpose.
- There has not been a lot of time for anyone to search for a short-cut solution to MPJ, nor the volume of sensitive material to motivate such a search as DES.

The main weaknesses of the MPJ encryption algorithm are:

- The key and block sizes are fixed (optimized for the memory and processing power of the IBM PC), so it doesn't take very good advantage of more powerful computers other than running faster.
- The software implementation isn't very fast.
- There is no way to prove that there is no short-cut solution to MPJ (or DES, or almost any encryption algorithm except for the one-time key tape).

VIII. DATA COMPRESSION

A. Purpose

Data compression has two obvious purposes: to save on communication channel capacity required to send a message and to save on media capacity required to store a message. The third purpose for data compression, pointed out by C. E. Shannon, is to make decryption of a message more difficult [SHA]. In reading through the history of cryptanalysis, it became very obvious to me that the one thing that all cryptanalysis

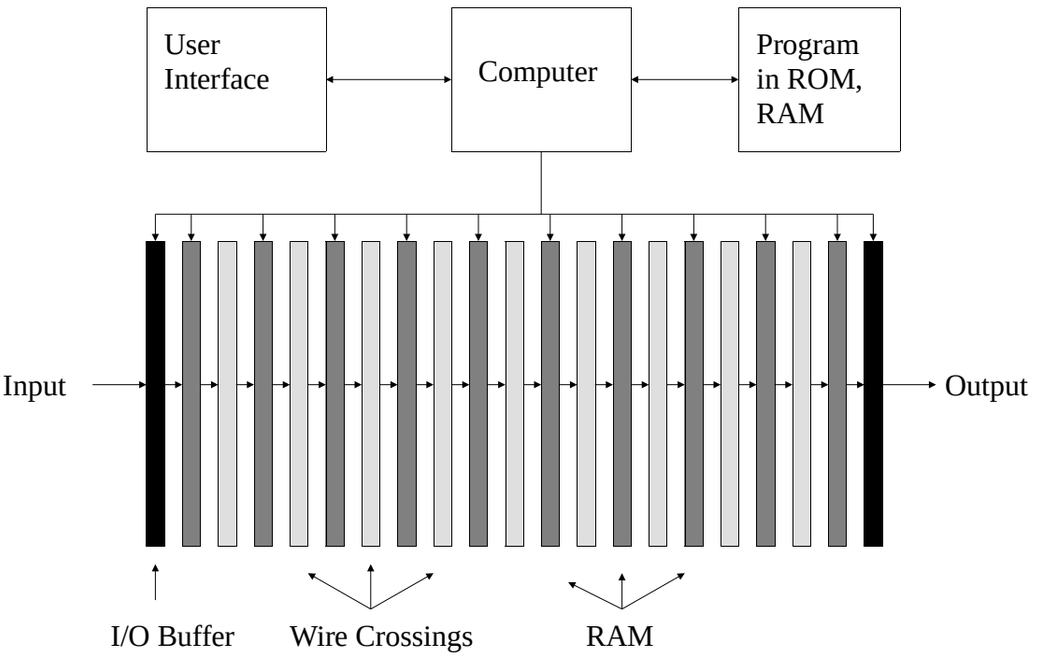


Figure VII.C. Hardware Implementation of MPJ

relies on the most is the presence of a great deal of redundancy present in natural languages. The more a cryptanalyst knows about the expected nature of a message, the more he can use this redundancy to his advantage. It is this redundancy that enables the cryptanalyst to tell which of several possible solutions is the right one — the one that ‘‘makes sense.’’ For example, if the message is between English-speaking parties, then the of the possible messages ‘‘ACCEPT THE PROPOSAL,’’ ‘‘A;LSDI FUPE ZAARED,’’ and ‘‘XXDKFJ DDDLKJDJFFZA,’’ the first message is instantly recognized as the only one that makes sense. The reason that this is so obvious is because of the redundancy of natural languages. If all of the redundancy of a message is removed, then there is no way to select one possible answer over another, and the cryptanalyst is left baffled.

The task at hand is to try to eliminate or at least reduce the amount of redundancy in a message before encryption, then restore the message to its original meaning after decryption. If all of the redundancy of a message is removed, it means that all possible messages that can be constructed from an alphabet have meaning, and that the length of a message is inversely proportional to its probability. This ideal can probably be achieved only for special circumstances with very limited messages. For use with general correspondence in a natural language, however, it is not practical to totally remove the redundancy from all messages. In fact, it is very difficult to even measure redundancy or entropy in a natural language. The concepts of redundancy and entropy (as defined in communication theory texts) are useful for comparisons of various methods of redundancy reduction.

Most existing methods of redundancy reduction use either a manually constructed code like the amateur radio operator's Q-signals or an automatic method that operates on either the alphabet or some groupings of alphabet symbols, like trigraphs. Manual techniques are fine for their intended application, but automatic methods that are well-suited for computer implementation are better for use with encryption. Automatic methods are likely to achieve good results with much greater ease of use and less opportunities for errors.

The probability of occurrence of various letters in most natural languages are fairly constant over a wide range of types of text, and have been studied and published many times. These probabilities for English letters are published in several of the texts on cryptography listed in the references section of this thesis. Probabilities of digraphs (groups of two letters) and trigraphs (groups of three letters) have likewise been studied, and are fairly constant for a given language. Just changing the representation of trigraphs to variable length codes that are shorter for more common trigraphs and longer for less common trigraphs results in a substantial savings in length of an English message.

There are many such methods for compressing data that are fairly straight forward. This and other methods, many of which are discussed by James Storer in his book on data compression [STO]. There are also several public domain and share-ware programs that are available on many bulletin boards that perform data compression. My favorite is PKARC, written by Phil Katz [KAT]. It dynamically analyzes each input file to determine which one of five compression methods or storing with no compression results in the smallest file. The compression methods used by PKARC are repetition coding,

Huffman encoding, Ziv-Lempel-Welch compression, and two different variations on Dynamic Ziv-Lempel-Welch compression. PKARC also has an option that allows a rather crude encryption of the archive files with a password. Although this doesn't provide strong cryptographic security, it does help to obscure the redundancy that PKARC adds back in, in the form of cyclic redundancy checks on files and file headers. The CRC checks and file headers would be a good point of vulnerability when trying to cryptanalyze an archived (compressed) file.

My feeling was that it was possible to compress natural language text even more than was commonly being done by using linguistic parsing of the text to form a dictionary. I was partially right. My method does remove more redundancy and make most text files smaller and more secure. It won't win any speed contests with PKARC, however, which I recommend that you use if speed is of great importance or if you are going to compress binary files (such as executable programs).

In comparing the performance of SQUEEZE with PKARC, I compressed a 4,316,030 byte ASCII text file (the King James Version of the Holy Bible) to 1,312,493 bytes with a language code file of 163,236 bytes. PKARC compressed the same file to 1,779,184 bytes. This is a removal of 69.6% of the redundancy for SQUEEZE (or 65.8% if you add the size of the language code file), compared with a redundancy reduction of only 58.8% for PKARC. This may not seem like much of a difference, but the smaller compressed file fits on a 1.44 Megabyte 3 1/2 inch floppy disk, and the other one doesn't. This would be more of a significant achievement if SQUEEZE weren't so slow. It took well over two hours to do this running on one of the fastest PC's available (25 MHz 80386),

while PKARC worked for about a minute on the task. I believe that both the speed and amount of redundancy reduction can be improved upon. I am therefore publishing the source code for what I did in Appendix B, so that someone else may be able to build upon this idea.

B. Linguistic Parsing

Instead of developing a Huffman code based on trigraphs or other such arbitrary chunks of fixed length, why not develop a Huffman code based on the way people read words? Letter combinations made from statistically properly distributed trigraphs will probably contain a lot of valid English words, but they will contain a lot of garbage, too. If, however, all symbols correspond to English words (or punctuation), then a collection of random symbols will look a lot more like English. Of course, the arrangement of words will probably not make sense in either group, but an intuitive analysis indicates that a code based on linguistic parsing of words will contain a lot less redundancy than will a fixed-length code.

To define linguistic symbols, I called any grouping of alphabetic symbols unbroken by non-alphabetic symbols a word symbol. I called any group of contiguous spaces a space symbol. I called a carriage return + line feed combination a new line symbol. Any other punctuation and numbers were called symbols themselves. There may be more efficient ways to split up text into symbols, since, for example, words are almost always followed by a space. I also counted upper and lower case words different symbols. It may have been more efficient to use a modifier symbol to indicate that the following word is

capitalized or written in all capitol letters. Of course, if all of the input text is all upper case, as would be the case for some telegraphic type communications systems, this is not a concern.

Once a linguistic symbol is detected in the input text, it is encoded with its corresponding Huffman code representation. This is then decoded at the other end of the communications link or after retrieving the data from storage, using the same Huffman code.

So that the Huffman code generation does not have to be done each time a file is compressed, an escape code that meant ‘‘the following symbol of ____ bits is not in the code, and is to be interpreted directly’’ is used. For these symbols, a simple suppression of the most significant bit is used, resulting in a savings of one out of eight bits.

C. Huffman Coding

Given a set of symbols and their associated probabilities of occurrence, a Huffman code for those symbols is constructed by generating a tree [STO]. Start with a set of symbols as individual nodes. While there are still separate nodes, combine the two nodes with the lowest probability (resolve ties arbitrarily) into a new node with a probability equal to the sum of the probabilities of the two nodes that were combined. Attach one of the nodes to the new node via a 1 and the other node via a 0. After all of the nodes are combined, the code for each symbol is the sequence of ones and zeroes assigned to the branches of the tree starting at the root and ending at the symbol.

Note that this algorithm results in a very similar code to a Shannon-Fano code (which generates the tree by starting at the root and building out to the leaves), but it is computationally more efficient to implement.

D. Pascal Programs

The pascal programs that implement this linguistic file compression are in appendix B. The first one, COUNT, merely counts the frequency of occurrence of words in sample text. It is not necessary that the text analyzed be the same text to be compressed, just that it be similar. For example, better compression would be expected when using a code based on personnel records when using that code to compress personnel records than when compressing medical discussions. However, the significant gains made in compressing the more common words in the code offset the lesser compression (or even possible expansion) encountered when words not in the code are encountered. It is also desirable to COUNT a large sample of text to ensure accurate word frequencies.

The second program, MAKETREE takes the frequency data from the output file generated by COUNT and generates a Huffman code from it. It then writes this tree out into a file that is used by SQUEEZE. To get reasonable speed from this program, it uses as much memory as MS-DOS will let it for the more frequently accessed information, and uses a temporary file for the rest of the information. The program runs significantly faster if this temporary file is on a RAM disk in extended or expanded memory (beyond the basic 640 KBytes of the MS-DOS domain).

SQUEEZE, the third program, does the squeezing and unsqueezing of files based on the output file of MAKETREE. The same Huffman coding tree can be used for many different input text files, because the statistics of English (especially when the same types of text, such as all business letters or all technical reports) remain fairly constant even when the content of the text conveys different meanings. SQUEEZE handles words not in its dictionary in stride, but highlights them as they scroll across the screen in processing so that the user can get a feel for how well the text is matched to the code tree in use. If only a few proper names and such are highlighted, then the code in use is a good one to use. If not, then perhaps a regeneration of the code would be in order. Note that the code file used to SQUEEZE a file must be identical to the one used to unSQUEEZE it, so it must either be stored or sent with the file, or sent in advance and agreed upon by communicating parties.

After testing these programs on various ASCII text files, it can be seen that significant compression can be obtained, and that the compressed files can be recovered reliably. There is, however, room for improvement in the exact way that text is parsed and in the speed of the implementation. Nevertheless, when maximum security is desired, I recommend first compressing an ASCII text file with SQUEEZE, then encrypting it with CRYPTMPJ. The code file (LANGUAGE.COD) used by SQUEEZE and the encryption key should be sent by a separate, secure channel to the receiving party (or stored in a physically secure location for data storage) in advance of the message. The combination of minimal natural redundancy in the message sent and the use of a good encryption algorithm should be difficult to crack.

IX. CONCLUSION

The MPJ Encryption algorithm provides an alternative to DES that is more secure, providing that the software and/or hardware does not get corrupted or tampered with, and providing that the keys are managed effectively. I have implemented and tested the algorithm in software for the IBM PC and compatible machines. The algorithm may be implemented directly in hardware for faster data rates. The MPJ Encryption Algorithm may be used by itself, or it may be improved upon by compressing the data before encryption.

The use of data compression in conjunction with any encryption algorithm drastically increases the security of the encrypted data. For natural language text, one approach that yields improved compression over the compression of constant size blocks of data is the compression of linguistic units, such as words. I have achieved reversible compression of such files to less than 35% of their original size using linguistic parsing and a Huffman code. For binary data, such as computer programs, the use of existing techniques, such as those in PKARC, written by Phil Katz and available on most computer bulletin boards, are recommended.

It is hoped that the MPJ encryption algorithm, as well as some of my ideas on data compression, will make a positive contribution towards data security, communications privacy, and efficiency of data storage and transmission. May God prosper those who use this knowledge and build upon it for good.

REFERENCES

[ALB] Douglas J. Albert and Stephen P. Morse,‘‘Combating Software Piracy by Encryption and Key Management,’’*Computer*, Vol. 21, No. 5, April 1984, pp. 68—73 Los Alamitos, CA: IEEE Computer Society.

[AME] Stanley R. Ames, Jr., Morrie Gasser, and Roger R. Schell, ‘‘Security Kernel Design and Implementation: An Introduction,’’ *Computer*, Vol. 16, No. 7, July 1983, pp. 14—22 Los Alamitos, CA: IEEE Computer Society.

[BAL] W. W. Rouse Ball & H. S. M. Coxeter, *Mathematical Recreations & Essays*. New York: The MacMillan Company, 1962.

[BAU] Friedrich L. Bauer, ‘‘Cryptology — Methods and Maxims,’’ in *Proceedings of the Workshop on Cryptography at Burg Feuerstein, Germany, March 29—April 2, 1982*, Berlin, Heidelberg, New York: Springer-Verlag, 1983.

[BEK] H. J. Beker, ‘‘A Survey of Encryption Algorithms,’’ in *International Conference on Secure Communication Systems*, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984. TK5102.5.I519.1984

[BET] Thomas Beth, Introduction to *Proceedings of the Workshop on Cryptography at Burg Feuerstein, Germany, March 29 — April 2, 1982*, Berlin, Heidelberg, New York: Springer-Verlag, 1983.

[BON] D. J. Bond, ‘‘Practical Primality Testing,’’ in *International Conference on Secure Communication Systems*, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984. TK5102.5.I519.1984

[BRO] C. B. Brookson and S. C. Serpell, ‘‘Security on the British Telecom Satstream Service’’ in *International Conference on Secure Communication Systems*, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984. TK5102.5.I519.1984

[BUR] Dave Bursky, ‘‘Protect Your EEPROM Data and Gain More Flexibility,’’ *Electronic Design*, 26 May 1988, pp.43—48. Waseca, MN: Brown Printing Co.

[CHA] Leslie S. Chalmers, ‘‘An Analysis of the Differences Between the Computer Security Practices in the Military and Private Sectors,’’ in *Proceedings of the IEEE 1986 Symposium on Security and Privacy*, Oakland, CA, April 7-9, 1986. QA76.9.A25.595.1986

[CHI] H. R. Chivers, ‘‘A Practical Fast Exponentiation Algorithm for Public Key,’’ in *International Conference on Secure Communication Systems*, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984. TK5102.5.I519.1984

[COO] R. H. Cooper, ‘‘Linear Transformations in Galois Fields and their Application to Cryptography,’’ *Cryptologia Magazine*, Volume 4, Number 3, July 1980, pages 184-188. Republished electronically by Tony Patti.

[COS] Terry Costlow, ‘‘Global Computer Network Cracks Cryptographic Mathematics Barrier,’’ *Electronic Engineering Times*, Issue 508, 17 October 1988, p. 14, Manhasset, NY: CMP Publications, Inc.

[CRC] *Handbook of Chemistry and Physics*, 53rd Edition, Cleveland, OH: The Chemical Rubber Company, 1972.

[DEA] Cipher A. Deavers and Louis Kruh, *Machine Cryptography and Modern Cryptanalysis*, Norwood, MA: Artech House, Inc., 1985. Z103.D43.1985

[DEN] Dorothy Elisabeth Robling Denning, *Cryptography and Data Security*, Reading, MA; Menlo Park, CA; London; Amsterdam; Don Mills, Ontario; Sydney: Addison-Wesley Publishing Company, 1982. QA76.9.A25.D46.1982

[DES] National Bureau of Standards, *Federal Information Processing Standards Publication Number 46* Dated 15 January 1977.

[DOH] Richard Doherty, ‘‘FBI Nabs Pirated VCII,’’ *Electronic Engineering Times, Manhasset, N. Y.: CMP Publications, Inc., Issue 500, August 22, 1988.*

[EET] ‘‘NSA OKs Decryption Unit,’’ *Electronic Engineering Times*, August 29, 1988, Manhasset, N. Y.: CMP Publications, Inc.

[FRA] Ole Immanuel Franksen, *Mr. Babbage’s Secret — The Tale of a Cipher and APL*, Englewood Cliffs, NJ: Prentice-Hall, 1985. Z103.B2.F72.1985

[FRI] H. J. Beker, J. M. K. Friend, and P. W. Halliden, ‘‘Simplifying Key Management in Electronic Fund Transfer Point of Sale Systems,’’ in *International Conference on Secure Communication Systems*, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984. TK5102.5.I519.1984

[FRM] Lester J. Fraim, ‘‘Scomp: A Solution to the Multilevel Security Problem,’’ *Computer*, Vol. 16, No. 7, July 1983,pp. 26—34, Los Alamitos, CA: IEEE Computer Society.

[GOO] G. Goos and J. Hartmans, *Lecture Notes in Computer Science*. Z102.5.W67.1982

[GOR] J. A. Gordon and H. Retkin, ‘‘Are Big S-Boxes Best?’’ in *Proceedings of the Workshop on Cryptography at Burg Feuerstein, Germany, March 29—April 2, 1982*, Berlin, Heidelberg, New York: Springer-Verlag, 1983.

[HAE] D. G. Haenshke, D. A. Kettler, and E. Oberer, ‘‘Network Management and Congestion in the U. S. Telecommunications Network,’’ IEEE Transactions on Communications, volume COM-29,pp. 376–385, April 1981.

[HEL] Gilbert Held and Thomas R. Marshall, *Data Compression*, Second Edition; Chichester, New York, Brisbane,Toronto, Singapore: John Wiley & Sons, 1987. QA76.9.D33.H44.1987

[HIL] Lester S. Hill, ‘‘Cryptography in an Algebraic Alphabet,’’ *American Mathematical Monthly*, June 1929, pages 306-312, the American Mathematical Society. Republished electronically by Tony Patti by permission.

[HLL] Lester S. Hill, ‘‘Concerning Certain Linear Transformation Apparatus of Cryptography,’’ *American Mathematical Society*, March 1931, pages 135-154. Republished electronically by Tony Patti by permission.

[JAC] A. M. Jackson, N. A. McEvoy, and B. B.Newman, ‘‘Project Universe Encryption Experiment’’ in *International Conference on Secure Communication Systems*, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984. TK5102.5.I519.1984

[KAH] David Kahn, *The Codebreakers — The Story of Secret Writing*, New York: The MacMillan Company, 1987. Z103.K28

[KRU] Cipher A. Deavers, David Kahn, Louis Kruh, Greg Mellen, and Brian Winkel, *Cryptology Yesterday, Today and Tomorrow*, Norwood, MA: Artech House, Inc., 1987. Z103.C76.1987

[LAN] Carl E. Landwehr, *The Best Available Technologies for Computer Security,” Computer*, Vol. 16, No. 7, July 1983,pp. 86—100, Los Alamitos, CA: IEEE Computer Society.

[KAT] Phil Katz, *PKARC FAST! Archive Create/Update Utility Version 3.5 04-27-87*, published electronically in PKX35A35.EXE.

[KON] Alan G. Konheim, *Cryptography: A Primer*, New York: John Wiley & Sons. Z103.K66

[KOZ] Wladyslaw Kozaczuk, *Enigma — How the German Machine Cipher was Broken and How it was Read by the Allies in World War Two*, University Publications of America, Inc., 1984.D810.C88.K6813.1984

[LIN] Christer Lindén, ‘‘Electronic Seal for Protection of Electronic Money in Sweden, Finland, and Norway’’ in *Proceedings of 1986 International Carnahan Conference on Security Technology: Electronic Crime Countermeasures*, August 12-14, 1986. QA76.9.A25.C41.1986

[LOD] N. Lodge, B. Flannaghan, and R. Morcom,‘‘Vision Scrambling of C-MAC DBS Signals,’’ in *International Conference on Secure Communication Systems*, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984. TK5102.5.I519.1984

[LU] W. P. Lu and M. K. Sandareshan, ‘‘A Hierarchical Key Management Scheme for End-to-End Encryption in Internet Environments’’ in *Proceedings of the IEEE 1986 Symposium on Security and Privacy*, Oakland, CA, April 7-9, 1986. QA76.9.A25.595.1986

[MCL] Vin McLellan, ‘‘Drugs and DES: A New Connection,’’ *Digital Review*, August 24, 1987.

[MER] Ralph C. Merkle, *Secrecy, Authentication, and Public Key Systems*, Ann Arbor, MI: UMI Research Press,1982. QA76.9.A25M47.1982

[MEY] Carl H. Meyer & Stephen M. Matyas,*Cryptography: a New Dimension in Computer Data Security — A Guide for the Design and Implementation of Secure Systems*, New York, Chichester, Brisbane, Toronto, Singapore: John Wiley & Sons, 1982. Z103.M55.1982

[MIT] C. J. Mitchell, ‘‘A Comparison of the Cryptographic Requirements for Digital Secure Speech Systems Operating at Different Bit Rates,’’ in *International Conference on Secure Communication Systems*, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984. TK5102.5.I519.1984

[NEW] David B. Newman, Jr. and Raymond L. Pickholtz, ‘‘Cryptography in the Private Sector,’’ *IEEE Communications Magazine*, August 1986, Volume 24, Number 8, pp. 7—10. New York, NY: The Institute of Electrical and Electronic Engineers, Inc.

[PAT] Tony Patti, *A Galois Field Cryptosystem*, 1986. Published electronically by Tony Patti, editor of Cryptosystems Journal, 9755 Oatley Lane, Burke, VA 22015.

[PIP] Fred Piper, ‘‘Stream Ciphers,’’ in *Proceedings of the Workshop on Cryptography at Burg Feuerstein, Germany, March 29—April 2, 1982*, Berlin, Heidelberg, New York: Springer-Verlag, 1983.

[PER] Tekla S. Perry and Paul Wallich, ‘‘Can Computer Crime be Stopped?,’’ *IEEE Spectrum*, May, 1984, pp.34—49. New York, NY: The Institute of Electrical and Electronic Engineers, Inc.

[RET] H. Retkin, ‘‘Multi-Level Knapsack Encryption,’’ in *International Conference on Secure Communication Systems*, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984. TK5102.5.I519.1984

[SAT] J. Sattler and C. P. Schnorr, ‘‘Ein Effizienzvergleich Der Faktorisierungsverfahren von Morrison-Brillhart und Schroeppel,’’ in *Proceedings of the Workshop on Cryptography at Burg Feuerstein, Germany, March 29—April 2, 1982*, Berlin, Heidelberg, New York: Springer-Verlag, 1983.

[SCH] C. P. Schnorr, ‘‘Is the RSA Scheme Safe?’’ in *Proceedings of the Workshop on Cryptography at Burg Feuerstein, Germany, March 29—April 2, 1982*, Berlin, Heidelberg, New York: Springer-Verlag, 1983.

[SHA] C. E. Shannon, ‘‘Communication Theory of Secrecy Systems,’’ *Bell System Technical Journal*, Volume 28, 1949.

[STO] James A. Storer, *Data Compression Methods and Theory*. Rockville, MD: Computer Science Press, 1988. QA76.9.D33S76

[TIL] Henk C. A. van Tilborg, *An Introduction to Cryptology*, Boston, Dordrecht, Lancaster: Kluwer Academic Publishers, 1988. Z103.T54.1987

[RET] H. Retkin, ‘‘Multi-Level Knapsack Encryption,’’ in *International Conference on Secure Communication Systems*, 22—23 February 1984, London, New York: Institute of Electrical Engineers, 1984. TK5102.5.I519.1984

[ULT] Ultron Labs Corporation, ‘‘Crypto Module Processes TOP SECRET Data,’’ EE Product News, October 1988, p. 16, Overland Park, KS: Intertec Publishing.

[VAN] J. Vandewalle, R. Govaerts, W. De Becker, M. Decroos, & G. Speybrouk, ‘‘RSA–Based Implementation of Public Key Cryptographic Protection in Office Systems’’ in *Proceedings of 1986 International Carnahan Conference on Security Technology: Electronic Crime Countermeasures*, August 12-14, 1986. QA76.9.A25.C41.1986

APPENDIX A. MPJ in Pascal

```
{$R-}    {Range checking off}
{$S-}    {Stack checking off}
{$I+}    {I/O checking on}
{$N-}    {Don't use 80x87 numeric coprocessor}
program cryptmpj;
{ Encryption designed to exceed DES in security value and be implementable on
  an IBM PC compatible machine. }

Uses Crt, Dos;  { Include screen handling & MS DOS interface functions. }

const maxinbuffer = 16;
      ap = #39;                        { Apostrophe for write statements. }

type blocks = array[0..15] of byte;            { 16 byte (128 bit) blocks. }
     stype = array[0..9,0..15,0..255] of byte;  { Holds substitution boxes. }
     ptrstype = ^stype;  { Dynamic variable required to get beyond 64K limit. }
     string80 = string[80];                { Long character strings. }
     string15 = string[15];                { Short character strings. }

var initial,                { Initialiazation vector. }
    feedback,               { Chaining feedback array. }
    key,                    { Encryption/decryption key. }
    buffer, outbuf:  blocks;  { File read/write buffers. }
    s,                      { Substitution boxes. }
    si: ptrstype;           { Inverse substitution boxes. }
```

```
    bytesdone,              { Number of bytes done. }
    number: longint;        { Used to read in key & initialization vector. }
    que,                    { Position in file specification que. }
    i, j, k,                { Iteration & array indexes. }
    actualread: integer;    { Actual number of bytes read in from file. }
    inputfile: file;        { File to encrypt or decrypt in place. }
    keyfile: text;          { External key file. }
    intkeyfile: file of stype; {Holds s and si. }
    keyname,                { Name of external key file. }
    intkeyname,             { Name of internal key file. }
    path,                   { Used in parsing file specification. }
    temp: string[255];      { Used in determining encrypt/decrypt option. }
    inputfilename,          { MS-DOS name of inputfile. }
    filespec: string80;     { File specification that may include wild cards}
    fileque: array[0..15] of string80; { Names of multiple file specifications}
    name: string15;         { Name of next file to process. }
    decryption,             { True iff decryption is desired. }
    filefound: boolean;     { At least one file was found to process.}
    ch: char;               { Character input. }
    search: searchrec;      { Used in finding matches to wild cards. }


procedure permute(x: blocks; var y: blocks);
{ This procedure is designed to make each bit of the output dependent on as
  many bytes of the input as possible, especially after repeated application.
  Each output byte takes its least significant bit from the corresponding
  input byte.  The next higher bit comes from the corresponding bit of the
```

input byte to the left. This is done until all bits of the output byte are filled. Where there is no byte to the left, the byte at the far right is used. }

```
begin
  y[0] := (x[0] and 1) or (x[1] and 2) or (x[2] and 4) or
          (x[3] and 8) or (x[4] and 16) or (x[5] and 32) or
          (x[6] and 64) or (x[7] and 128);
  y[1] := (x[1] and 1) or (x[2] and 2) or (x[3] and 4) or
          (x[4] and 8) or (x[5] and 16) or (x[6] and 32) or
          (x[7] and 64) or (x[8] and 128);
  y[2] := (x[2] and 1) or (x[3] and 2) or (x[4] and 4) or
          (x[5] and 8) or (x[6] and 16) or (x[7] and 32) or
          (x[8] and 64) or (x[9] and 128);
  y[3] := (x[3] and 1) or (x[4] and 2) or (x[5] and 4) or
          (x[6] and 8) or (x[7] and 16) or (x[8] and 32) or
          (x[9] and 64) or (x[10] and 128);
  y[4] := (x[4] and 1) or (x[5] and 2) or (x[6] and 4) or
          (x[7] and 8) or (x[8] and 16) or (x[9] and 32) or
          (x[10] and 64) or (x[11] and 128);
  y[5] := (x[5] and 1) or (x[6] and 2) or (x[7] and 4) or
          (x[8] and 8) or (x[9] and 16) or (x[10] and 32) or
          (x[11] and 64) or (x[12] and 128);
  y[6] := (x[6] and 1) or (x[7] and 2) or (x[8] and 4) or
          (x[9] and 8) or (x[10] and 16) or (x[11] and 32) or
          (x[12] and 64) or (x[13] and 128);
```

```
y[7] := (x[7] and 1) or (x[8] and 2) or (x[9] and 4) or
        (x[10] and 8) or (x[11] and 16) or (x[12] and 32) or
        (x[13] and 64) or (x[14] and 128);
y[8] := (x[8] and 1) or (x[9] and 2) or (x[10] and 4) or
        (x[11] and 8) or (x[12] and 16) or (x[13] and 32) or
        (x[14] and 64) or (x[15] and 128);
y[9] := (x[9] and 1) or (x[10] and 2) or (x[11] and 4) or
        (x[12] and 8) or (x[13] and 16) or (x[14] and 32) or
        (x[15] and 64) or (x[0] and 128);
y[10] := (x[10] and 1) or (x[11] and 2) or (x[12] and 4) or
        (x[13] and 8) or (x[14] and 16) or (x[15] and 32) or
        (x[0] and 64) or (x[1] and 128);
y[11] := (x[11] and 1) or (x[12] and 2) or (x[13] and 4) or
        (x[14] and 8) or (x[15] and 16) or (x[0] and 32) or
        (x[1] and 64) or (x[2] and 128);
y[12] := (x[12] and 1) or (x[13] and 2) or (x[14] and 4) or
        (x[15] and 8) or (x[0] and 16) or (x[1] and 32) or
        (x[2] and 64) or (x[3] and 128);
y[13] := (x[13] and 1) or (x[14] and 2) or (x[15] and 4) or
        (x[0] and 8) or (x[1] and 16) or (x[2] and 32) or
        (x[3] and 64) or (x[4] and 128);
y[14] := (x[14] and 1) or (x[15] and 2) or (x[0] and 4) or
        (x[1] and 8) or (x[2] and 16) or (x[3] and 32) or
        (x[4] and 64) or (x[5] and 128);
y[15] := (x[15] and 1) or (x[0] and 2) or (x[1] and 4) or
        (x[2] and 8) or (x[3] and 16) or (x[4] and 32) or
```

```
        (x[5] and 64) or (x[6] and 128);
  end;

procedure ipermute(x: blocks; var y: blocks);
{ This is the inverse of the procedure permute. }

  begin
    y[0] := (x[0] and 1) or (x[15] and 2) or (x[14] and 4) or
        (x[13] and 8) or (x[12] and 16) or (x[11] and 32) or
        (x[10] and 64) or (x[9] and 128);
    y[1] := (x[1] and 1) or (x[0] and 2) or (x[15] and 4) or
        (x[14] and 8) or (x[13] and 16) or (x[12] and 32) or
        (x[11] and 64) or (x[10] and 128);
    y[2] := (x[2] and 1) or (x[1] and 2) or (x[0] and 4) or
        (x[15] and 8) or (x[14] and 16) or (x[13] and 32) or
        (x[12] and 64) or (x[11] and 128);
    y[3] := (x[3] and 1) or (x[2] and 2) or (x[1] and 4) or
        (x[0] and 8) or (x[15] and 16) or (x[14] and 32) or
        (x[13] and 64) or (x[12] and 128);
    y[4] := (x[4] and 1) or (x[3] and 2) or (x[2] and 4) or
        (x[1] and 8) or (x[0] and 16) or (x[15] and 32) or
        (x[14] and 64) or (x[13] and 128);
    y[5] := (x[5] and 1) or (x[4] and 2) or (x[3] and 4) or
        (x[2] and 8) or (x[1] and 16) or (x[0] and 32) or
        (x[15] and 64) or (x[14] and 128);
    y[6] := (x[6] and 1) or (x[5] and 2) or (x[4] and 4) or
```

```
        (x[3] and 8) or (x[2] and 16) or (x[1] and 32) or
        (x[0] and 64) or (x[15] and 128);
y[7] := (x[7] and 1) or (x[6] and 2) or (x[5] and 4) or
        (x[4] and 8) or (x[3] and 16) or (x[2] and 32) or
        (x[1] and 64) or (x[0] and 128);
y[8] := (x[8] and 1) or (x[7] and 2) or (x[6] and 4) or
        (x[5] and 8) or (x[4] and 16) or (x[3] and 32) or
        (x[2] and 64) or (x[1] and 128);
y[9] := (x[9] and 1) or (x[8] and 2) or (x[7] and 4) or
        (x[6] and 8) or (x[5] and 16) or (x[4] and 32) or
        (x[3] and 64) or (x[2] and 128);
y[10] := (x[10] and 1) or (x[9] and 2) or (x[8] and 4) or
        (x[7] and 8) or (x[6] and 16) or (x[5] and 32) or
        (x[4] and 64) or (x[3] and 128);
y[11] := (x[11] and 1) or (x[10] and 2) or (x[9] and 4) or
        (x[8] and 8) or (x[7] and 16) or (x[6] and 32) or
        (x[5] and 64) or (x[4] and 128);
y[12] := (x[12] and 1) or (x[11] and 2) or (x[10] and 4) or
        (x[9] and 8) or (x[8] and 16) or (x[7] and 32) or
        (x[6] and 64) or (x[5] and 128);
y[13] := (x[13] and 1) or (x[12] and 2) or (x[11] and 4) or
        (x[10] and 8) or (x[9] and 16) or (x[8] and 32) or
        (x[7] and 64) or (x[6] and 128);
y[14] := (x[14] and 1) or (x[13] and 2) or (x[12] and 4) or
        (x[11] and 8) or (x[10] and 16) or (x[9] and 32) or
        (x[8] and 64) or (x[7] and 128);
```

```
   y[15] := (x[15] and 1) or (x[14] and 2) or (x[13] and 4) or
            (x[12] and 8) or (x[11] and 16) or (x[10] and 32) or
            (x[9] and 64) or (x[8] and 128);
  end;


procedure makesbox(key: blocks);
{ This procedure generates internal keys by filling the substitution box array
  s^ based on the external key given as input. }
  var i, j, k: integer;
  procedure makeonebox(i, j: integer; key: blocks);
    var pos, m, n, p, startbit, bitmask, startbyte, keybyte: word;
        empty: array[0..255] of boolean;
    begin
      for m := 0 to 255 do    { The empty array is used to make sure that }
        empty[m] := true;     { each byte of the array is filled only once. }
      startbit := 1;
      startbyte := 0;
      for n := 255 downto 128 do  { n counts the number of bytes left to fill }
        begin
          keybyte := startbyte;
          bitmask := startbit;
          m := 0;
          for p := 0 to 7 do   { m is obtained by bit selection on the key }
            begin
              m := m or (key[keybyte] and bitmask);
              bitmask := bitmask shl 1;
```

```
    if bitmask > 128 then
     begin
      bitmask := 1;
      inc(keybyte);
      if keybyte > 15 then keybyte := 0;
     end;
  end;
 pos := (n * m) div 255;  { pos is the position among the UNFILLED }
 m := 0;                  { components of the s^ array that the    }
 p := 0;                 { number n should be placed.  }
 while m < pos do
  begin
   inc(p);
   if empty[p] then inc(m);
  end;
 while not empty[p] do inc(p);
 s^[i, j, p] := n;
 empty[p] := false;
 startbit := startbit shl 1;  { The starting position of the bit  }
 if startbit > 128 then       { selection for the key is rotated  }
  begin                    { left one bit for the next n.      }
   startbit := 1;
   inc(startbyte);
   if startbyte > 15 then startbyte := 0
  end;
end;
```

```
startbyte := 0;
startbit := 1;
for n := 127 downto 1 do        { This half of the algorithm is the   }
 begin                          { same as the upper half, except that }
  keybyte := startbyte;         { only 7 bits are selected for m.     }
  bitmask := startbit;
  m := 0;
  for p := 0 to 6 do
   begin
    m := m or (key[keybyte] and bitmask);
    bitmask := bitmask shl 1;
    if bitmask > 64 then
     begin
      bitmask := 1;
      keybyte := keybyte + 3;
      if keybyte > 15 then keybyte := keybyte - 16;
     end;
   end;
  pos := (n * m) div 127;
  m := 0;
  p := 0;
  while m < pos do
   begin
    inc(p);
    if empty[p] then inc(m);
   end;
```

```
      while not empty[p] do inc(p);
      s^[i, j, p] := n;
      empty[p] := false;
      startbit := startbit shl 1;
      if startbit > 64 then
        begin
          startbit := 1;
          inc(startbyte);
          if startbyte > 15 then startbyte := 0
        end;
    end;
  p := 0;
  while not empty[p] do
    inc(p);
  s^[i,j,p] := 0;
end;


begin
  new(s);
  for i := 0 to 9 do
    for j := 0 to 15 do
      begin
        write(#13,'Filling substitution boxes for round ',i+1,' of 10, byte ',
              j+1,' of 16.  ');
        makeonebox(i, j, key);
        permute(key, key);            { Shuffle key bit positions.        }
```

```
        for k := 0 to 15 do          { Run key through last s-box before }
          key[k] := s^[i, j, key[k]]; { making the next s-box.           }
      end;
   writeln;
 end;


procedure makesi;
{ This procedure fills the inverse substitution box array si^.  It is not
  necessary to call this procedure unless the decryption mode is used.  }
  var i, j, k: integer;
  begin
    new(si);
    for i := 0 to 9 do
      for j := 0 to 15 do
        for k := 0 to 255 do
          si^[i, j, s^[i, j, k]] := k;
  end;


{$I CRYPTINC.PAS}    { Include additional file handling & user interface. }


procedure substitute(round: integer; x: blocks; var y: blocks);
var i: integer;
begin
  for i := 0 to 15 do
    y[i] := s^[round,i,x[i]];
end;
```

```
procedure isubst(round: integer; x: blocks; var y: blocks);
var i: integer;
begin
  for i := 0 to 15 do
    y[i] := si^[round,i,x[i]];
end;

procedure encrypt(x: blocks; var y: blocks);
{ Encrypt a block of 16 bytes. }
var z: blocks;
begin
  substitute(0, x, y);
  permute(y, z);
  substitute(1, z, y);
  permute(y, z);
  substitute(2, z, y);
  permute(y, z);
  substitute(3, z, y);
  permute(y, z);
  substitute(4, z, y);
  permute(y, z);
  substitute(5, z, y);
  permute(y, z);
  substitute(6, z, y);
  permute(y, z);
```

```
  substitute(7, z, y);
  permute(y, z);
  substitute(8, z, y);
  permute(y, z);
  substitute(9, z, y);
end;

procedure decrypt(x: blocks; var y: blocks);
{ Decrypt a block of 16 bytes. }
var z: blocks;
begin
  isubst(9, x, y);
  ipermute(y, z);
  isubst(8, z, y);
  ipermute(y, z);
  isubst(7, z, y);
  ipermute(y, z);
  isubst(6, z, y);
  ipermute(y, z);
  isubst(5, z, y);
  ipermute(y, z);
  isubst(4, z, y);
  ipermute(y, z);
  isubst(3, z, y);
  ipermute(y, z);
  isubst(2, z, y);
```

```
  ipermute(y, z);
  isubst(1, z, y);
  ipermute(y, z);
  isubst(0, z, y);
end;

begin
  startup;  { Initialize files, call makesbox (and makesi if required). }
  que := 0;
  while (que < 16) and (fileque[que] <> ’’) do
    begin
      filespec := fileque[que];
      inputfilename := filespec;
      path := ’’;
      repeat
        i := pos(’\’,inputfilename);
        if i > 0 then
          begin
            path := path + copy(inputfilename, 1, i);
            inputfilename := copy(inputfilename, i+1, length(inputfilename));
          end;
      until i = 0;
      findfirst(filespec, $22, search);
      while doserror = 0 do
        begin
          inputfilename := path + search.name;
```

```
assign(inputfile, inputfilename);
reset(inputfile,1);
filefound := true;
bytesdone := 0;
for i := 0 to 15 do
  feedback[i] := initial[i];
if decryption then
 begin
  writeln('Decrypting ',inputfilename);
  while not eof(inputfile) do
    begin
      blockread(inputfile,buffer,maxinbuffer,actualread);
      encrypt(feedback,outbuf);
      { Note that decrypt is not ever called when using block
        chaining with ciphertext feedback.  It would be called
        if the electronic codebook mode were used, instead. }
      for i := 0 to 15 do
        begin
          outbuf[i] := outbuf[i] xor buffer[i];
          feedback[i] := buffer[i]
        end;
      seek(inputfile, filepos(inputfile) - actualread);
      blockwrite(inputfile, outbuf, actualread);
      bytesdone := bytesdone + actualread;
      write(#13, bytesdone:9,' bytes decrypted. ');
    end;
```

```
        writeln(#13, bytesdone:9,' bytes decrypted.  ');
       end
      else
       begin
        writeln('Encrypting ',inputfilename);
        while not eof(inputfile) do
          begin
            blockread(inputfile,buffer,maxinbuffer,actualread);
            encrypt(feedback,outbuf);
            for i := 0 to 15 do
              begin
                outbuf[i] := outbuf[i] xor buffer[i];
                feedback[i] := outbuf[i]
              end;
            seek(inputfile, filepos(inputfile) - actualread);
            blockwrite(inputfile, outbuf, actualread);
            bytesdone := bytesdone + actualread;
            write(#13, bytesdone:9,' bytes encrypted.  ');
          end;
        writeln(#13, bytesdone:9,' bytes encrypted.  ');
       end;
      close(inputfile);
      findnext(search);
    end;
  inc(que);
end;
```

```
  if not filefound then writeln('No matching files found.');
  writeln('Done.');
end.
```

APPENDIX B. Linguistic Data Compression Programs

```
program count;

{Program to count the probability of occurances of words in text.}


uses dos, crt;

const maxbuf = 4096;

      maxstr = 15;

      ap = #39;


type wd = string[maxstr];

     ptr = ^entry;

     yarn = string[80];

     entry = record

               wrd:  wd;        {Linguistic word}

               count: longint;  {How many times this word was found}

               next: ptr;       {Next record in sequence}

             end;


var outfile,             {File with output report}

    infile: text;        {File with input text to be analyzed}

    bottom,              {Last entry record}

    newbuf,              {New entry record}

    prev: ptr;           {Previous entry record}

    buffer,              {Current entry record}

    top: array[1..255] of ptr;  {First entry record}

    words: wd;           {Word under construction}
```

```
   temp,
   path,                {Path of input file}
   filemask,            {File name mask}
   filespec,            {Path + name (including wild cards)}
   inname,              {Name of input file}
   outname: yarn;       {Name of output file}
   ch: char;
   i, j, cmdparameter, topindex: integer;
   cnt: longint;
   filefound: boolean;
   inbuf, outbuf: array[1..maxbuf] of byte;
   s: searchrec;        {Record fill: array[1..21] of byte;attr:byte;time,
                        size:longint;name: string[12]}

function exist(filename: wd): boolean;
{Returns TRUE iff the file exists.}
var f: file;
begin {exist}
  assign(f,filename);
  {$I-}
  reset(f);
  {$I+}
  if IOresult = 0 then
    begin
      exist := true;
      close(f)
```

```
    end
  else
    exist := false;
end;  {exist}


function punct(ch: char): boolean;
{TRUE iff ch is not a letter or chr(255).}
  var c:  integer;
  begin
    c := ord(ch);
    if ((c > 10) and (c <= 64)) or ((c >= 91) and (c <= 96))
    or ((c >= 123) and (c <= 254)) or ((c > 0) and (c < 10)) then
      punct := true
    else
      punct := false;
  end;


function letter(ch: char): boolean;
{TRUE iff ch is a letter A-Z or a-z.}
  var c: integer;
  begin
    c := ord(ch);
    if ((c >= 65) and (c <= 90)) or ((c >= 97) and (c <= 122)) then
      letter := true
    else
      letter := false;
```

```
  end;


procedure parse(var words: wd; var ch: char);
{Extracts a word or item of punctuation from input file.}
  begin  {parse}
    words := '';
    while not (letter(ch) or punct(ch) or eof(infile)) do
      begin
        read(infile,ch);
      end;
    if punct(ch) then
      begin
        if ord(ch) = 13 then
          begin
            words := chr(255)+chr(255);
            if not eof(infile) then read(infile, ch);
          end
        else
          if ch = ' ' then
            begin
              words := ch;
              while (not eof(infile)) and (ch = ' ') and (length(words) < maxstr) do
                begin
                  read(infile,ch);
                  if ch = ' ' then words := words + ch;
                end;
```

```
          if (length(words) = maxstr) and (not eof(infile)) then
            read(infile,ch);
          end
        else
          begin
            words := ch; {return one punctuation character}
            if not eof(infile) then read(infile, ch);
          end;
    end
  else
    begin
      if letter(ch) then
        begin
          words := ch;
          while (not eof(infile)) and letter(ch) and (length(words) < maxstr) do
            begin
              read(infile,ch);
              if letter(ch) then words := words + ch;
            end;
          if (length(words) = maxstr) and (not eof(infile)) then
            read(infile,ch);
        end;
    end;
 end;  {parse}


procedure update(words: wd);
```

```
{Increments the count associated with a word.}
 begin
   topindex := ord(words[1]);
   buffer[topindex] := top[topindex];
   {Find record that matches word, if it exists.}
   while (buffer[topindex]^.wrd < words) and (buffer[topindex]^.next <> nil) do
     begin
       prev := buffer[topindex];
       buffer[topindex] := buffer[topindex]^.next
     end;

   if buffer[topindex]^.wrd = words then
     {Increment count on word that already exists}
     inc(buffer[topindex]^.count)
   else
     begin  {Add word to middle of list with count of one}
       new(newbuf);
       newbuf^.wrd := words;
       newbuf^.count := 1;
       newbuf^.next := prev^.next;
       prev^.next := newbuf;
     end;
 end;

procedure report;
{Produces a report file.}
```

```
 var total: longint;
 begin
  assign(outfile, paramstr(1));
  rewrite(outfile);
  total := 0;
  for topindex := 1 to 255 do
   begin
    buffer[topindex] := top[topindex];
    while (buffer[topindex]^.next <> nil) do
     begin
      buffer[topindex] := buffer[topindex]^.next;
      if buffer[topindex]^.count > 0 then
       begin
        writeln(outfile,buffer[topindex]^.wrd);
        writeln(outfile,buffer[topindex]^.count);
        total := total + buffer[topindex]^.count;
       end;
     end;
   end;
  writeln(outfile, chr(255));
  writeln(outfile, '1');
  writeln(outfile, 'Total count:  ',total);
  close(outfile);
 end; {report}


begin
```

```
clrscr;
writeln('  This program counts words in a set    This program may be copied freely ');
writeln('  of input files to determine their     for educational use or to try it  ');
writeln('  frequency of occurance.  This data    out.  If you like it, donations   ');
writeln('  is then stored in a file for use by   will be accepted by the author.   ');
writeln('  MAKETREE, which constructs a code     This program is believed to be    ');
writeln('  tree with the Huffman algorithm.      reliable, but it is the user',ap,'s    ');
writeln('  The code tree is used by SQUEEZE.     responsibility to determine its   ');
writeln('                                 fitness for use.  No liability    ');
writeln('  Mike Johnson                        will be assumed by the author.    ');
writeln('  P. O. Box 1151                      Copyright (C) 1988 Mike Johnson.  ');
writeln('  Longmont, CO 80502-1151             All rights reserved.              ');
if paramcount < 2 then
 begin
   writeln;
   writeln('Syntax: COUNT outputfilename inputfilename[s]');
   writeln('       input file names may include * and ?');
   writeln('       Input files are assumed to be ASCII text.');
   writeln;
   writeln('If a previous COUNT ouput file named LANGUAGE.OUT exists, then new counts');
   writeln('will be added to the counts in LANGUAGE.OUT.');
   writeln('Maximum effectiveness will be obtained if the nature of the files COUNTed');
   writeln('is similar to the nature of the files to be compressed with SQUEEZE.');
 end
else
 begin
```

```
filefound := false;
new(bottom);
bottom^.wrd := chr(255);
bottom^.count := 0;
bottom^.next := nil;
for topindex := 1 to 255 do
 begin
   new(top[topindex]);
   top[topindex]^.wrd := chr(1);
   top[topindex]^.count := 0;
   top[topindex]^.next := bottom;
   buffer[topindex] := top[topindex];
 end;
ch := chr(0);
words := '                         ';

if exist('LANGUAGE.OUT') then
 begin
   assign(infile,'LANGUAGE.OUT');
   reset(infile);
   writeln('Reading in existing LANGUAGE.OUT.');
   while (not eof(infile)) and (words <> chr(255)) do
     begin
       readln(infile,words);
       if words <> chr(255) then
         begin
```

```
            topindex := ord(words[1]);
            readln(infile,cnt);
            new(newbuf);
            newbuf^.wrd := words;
            newbuf^.count := cnt;
            newbuf^.next := bottom;
            buffer[topindex]^.next := newbuf;
            buffer[topindex] := newbuf;
            write(#13,words,'           ');
          end;
      end;
    close(infile);
    writeln;
  end;

{Initialize input files.}
for cmdparameter := 2 to paramcount do
  begin
    filespec := paramstr(cmdparameter);
    filemask := filespec;
    path := '';
    repeat
      i := pos('\',filemask);
      if i > 0 then
        begin
          path := path + copy(filemask, 1, i);
```

```
            filemask := copy(filemask, i+1, length(filemask));
          end;
        until i = 0;
        findfirst(filespec, readonly + hidden + archive, s);
        while doserror = 0 do
         begin
          filefound := true;
          inname := path + s.name;
          assign(infile, inname);
          reset(infile);
          words := '                         ';
writeln('>>>>>>>>>>>>>>>>>>>> Processing ',inname);
          while (not eof(infile)) and (words <> '') and (not keypressed) do
           begin
            parse(words,ch);     {Find word or punctuation}
            if words = chr(255)+chr(255) then
             writeln
            else
             write(words);
            if length(words) > 0 then
             update(words);     {Increment word counter}
           end;
          close(infile);
          findnext(s);
         end;
       end;
```

```
    if not filefound then writeln('No matching files found.');
    report;
  end;
end.
```

```
{$R+}
{$M 3072,4096,655360}
program maketree;

uses crt;
const maxstr = 15;
      ap = #39;

type wd = string[maxstr];
    yarn = string[80];
    mem = ^mementry;
    entry = record
              wrd:  wd;        {Which linguistic word in input file}
              memory: mem;     {Corresponding mementry.}
            end;

    mementry = record
                 count: longint; {Word count.}
                 onezero: byte;  {Is this record pointed to by 1 or 0?}
                 root,           {Position of record nearer root.}
                 next:  mem;     {Position of next record.}
               end;

var temp: file of entry; {Temporary file}
   infile: text;        {File with input text to be analyzed}
   buffer,              {Current entry record}
```

```
    newbuf: entry;       {New entry record}
    mtop,
    mbottom,
    mbuffer,
    mnewbuf,
    mprev: mem;
    inname,              {Name of input file}
    outname,             {Name of output file}
    words: wd;           {Word under construction}
    ch: char;
    cnt1,
    cnt2,
    cnt:  longint;
    i, j: integer;
    tempname: yarn;

function exist(filename: wd): boolean;
{Returns TRUE iff the file exists.}
var f: file;
begin {exist}
  assign(f,filename);
  {$I-}
  reset(f);
  {$I+}
  if IOresult = 0 then
    begin
```

```
      exist := true;
      close(f)
    end
  else
    exist := false;
end;  {exist}

procedure fanno;
  begin
    cnt1 := 0;
    repeat

      {Find smallest two records.}
      mprev := mtop^.next;
      mbuffer := mprev^.next;

      {Combine lowest 2 probabilities of counts.}
      mtop^.next := mbuffer^.next;
      new(mnewbuf);
      mnewbuf^.count := mbuffer^.count + mprev^.count;
      mnewbuf^.onezero := 2;
      mnewbuf^.root := nil;
      mprev^.root := mnewbuf;
      mprev^.onezero := 1;
      mbuffer^.root := mnewbuf;
      mbuffer^.onezero := 0;
```

```
      {Place combined entry in proper place in chain.}
      mbuffer := mtop;
      repeat
        mprev := mbuffer;
        mbuffer := mbuffer^.next;
      until (mnewbuf^.count <= mbuffer^.count);
      mprev^.next := mnewbuf;
      mnewbuf^.next := mbuffer;

      {Check for completion.}
      inc(cnt1);
      write(#13,cnt1,' nodes combined.  ');
    until (cnt1 >= cnt2);
  end;  {fanno}

procedure placeit;
  begin
      write(#13,cnt:11,' ',words,'          ');
      mbuffer := mtop;
      repeat
        mprev := mbuffer;
        mbuffer := mbuffer^.next;
      until mbuffer^.count >= cnt;
      new(mnewbuf);
      newbuf.wrd := words;
```

```
        newbuf.memory := mnewbuf;
             write(temp,newbuf);
        mnewbuf^.count := cnt;
        mnewbuf^.onezero := 2;
             mnewbuf^.root := nil;
        mnewbuf^.next := mbuffer;
        mprev^.next := mnewbuf;
  end;

procedure filegen;
var cod: file of byte;
    ascnum: yarn;
    c: char;
    b, d: byte;
    i, shifter: integer;

  begin
    assign(cod,outname);
    rewrite(cod);
    reset(temp);
    repeat
      read(temp,buffer);
      mbuffer := buffer.memory;
      if buffer.wrd <> '' then
             begin
          for i := 0 to length(buffer.wrd) do
```

```
  begin
    b := ord(buffer.wrd[i]);
    write(cod,b);
  end;
ascnum := '';
while (mbuffer^.root <> nil) do
  begin
    if mbuffer^.onezero = 0 then
      ascnum := ascnum + '0'
    else
      ascnum := ascnum + '1';
    mbuffer := mbuffer^.root;
  end;
d := ord(ascnum[0]);
write(cod,d);
shifter := 1;
b := 0;
for i := d downto 1 do
  begin
    if ascnum[i] = '1' then b := b + shifter;
    shifter := shifter shl 1;
    if shifter >= 256 then
      begin
        write(cod,b);
        b := 0;
        shifter := 1
```

```
            end;
          end;
        if shifter > 1 then write(cod,b);
        write(#13,buffer.wrd:maxstr,' = ',ascnum,'                    ');
      end;
  until eof(temp);
  close(cod);
  close(temp);
  erase(temp);
end;

begin
  clrscr;
  writeln('  MAKETREE takes the word count        This program may be copied freely ');
  writeln('  data created by COUNT and creates    for educational use or to try it  ');
  writeln('  a code tree using the Huffman        out.  If you like it, donations   ');
  writeln('  algorithm.  The resulting code tree  will be accepted by the author.   ');
  writeln('  is used by SQUEEZE when it           This program is believed to be    ');
  writeln('  compresses ASCII text files.         reliable, but it is the user',ap,'s    ');
  writeln('                                       responsibility to determine its   ');
  writeln('                                       fitness for use.  No liability    ');
  writeln('  Mike Johnson                         will be assumed by the author.    ');
  writeln('  P. O. Box 1151                       Copyright (C) 1988 Mike Johnson.  ');
  writeln('  Longmont, CO 80502-1151              All rights reserved.              ');
  if paramcount < 2 then
    begin
```

```
    writeln('Syntax: MAKETREE infile outfile [tempfile]');
    writeln('               infile is input file name (generated by COUNT)');
    writeln('               outfile is output file name to be used by SQUEEZE');
    writeln('               tempfile is filename to use for temporary file');
    writeln('The temporary file is best put on a RAM disk in extended or expanded memory,');
    writeln('but a hard disk or even a floppy will do if that is the fastest you have.');
    writeln('Don',ap,'t use a RAM disk in conventional (lower 640K) memory, as this program');
    writeln('already uses most of that in generating the coding tree.');
  end
else
  begin
    inname := paramstr(1);
    outname := paramstr(2);
    if exist(inname) then
      begin
        assign(infile,inname);
        reset(infile);
      end
    else
      begin
        writeln('Unable to open ',inname);
        exit
      end;
    if paramcount > 2 then
      tempname := paramstr(3)
    else
```

```
 tempname := 'ZYXWVUTS.$$$';
assign(temp,tempname);
{$I- }
rewrite(temp);
{$I+ }
if ioresult <> 0 then
 begin
   writeln('I/O error attempting to open temporary file ',tempname);
   halt;
 end;
new(mtop);
new(mbottom);
mtop^.count := -1;
mtop^.onezero := 2;
mtop^.root := nil;
mtop^.next := mbottom;
mbottom^.count := 2147483647;
mbottom^.onezero := 2;
mbottom^.root := nil;
mbottom^.next := nil;
cnt2 := 2;
writeln('Reading in and sorting word frequency data.');
repeat
 readln(infile,words);
 readln(infile,cnt);
 placeit;
```

```
        inc(cnt2);
      until (words = chr(255)) or eof(infile);
      close(infile);
      words := chr(255) + chr(255);
      placeit;
      words := words + chr(255);
      placeit;
      words := words + chr(255);
      placeit;
      writeln(#13,cnt2,' nodes to add to Huffman coding tree.                ');
      fanno;
      writeln;
      writeln('Writing output file.');
      filegen;
      writeln(#13,'Done.                                    ');
    end;
end.
```

```
{$R+}
{$M 2048,4096,655360}
program squash;

uses dos, crt;

const maxbuf = 4096;
      maxstr = 15;
      ap = #39;

type wd = string[maxstr];
     yarn = string[80];
     ptr = ^entry;
     entry = record
               wrd:  wd;        {Linguistic word}
               count:  byte;    {How many bits in code}
               code: longint;   {Code (right justified in 4 byte integer)}
               next: ptr;       {Pointer to next entry}
             end;

var outfile,                {File with output report}
    infile: file;           {File with input text to be analyzed}
    temp,
    path,                   {Path of input file}
    filemask,               {File name mask}
    filespec,               {Path + name (including wild cards)}
```

```
inname,                 {Name of input file}
outname: yarn;          {Name of output file}
words: wd;              {Word under construction}
ch: char;
top,
bottom: array[1..32] of ptr;
last,
newbuf,
newline,
endfile,
buf: ptr;
mask,
masked,
place,
long,
cd,
cnt:  longint;
infilepos,              {Current position in input file buffer}
inbitpos,               {Current bit mask in input file buffer}
inbytes,                {Actual number of bytes read}
outfilepos,             {Current position in output file buffer}
outbitpos,              {Current bit mask in output file buffer}
outbytes:  word;        {Actual number of bytes to write}
cmdparameter,           {Command line parameter number}
start,                  {Starting point for command line file name scan}
strlen,
```

```
    bite,
    b, i, j, k, m: integer;
    squish, filefound, done: boolean;
    outbite:  byte;
    inbuf,
    outbuf: array[1..maxbuf] of byte;
    s: searchrec;          {Record fill: array[1..21] of byte;attr:byte;time,
                           size:longint;name: string[12]}


function exist(filename: yarn): boolean;
{Returns TRUE iff the file exists.}
var f: file;
begin {exist}
  assign(f,filename);
  {$I-}
  reset(f);
  {$I+}
  if IOresult = 0 then
    begin
      exist := true;
      close(f)
    end
  else
    exist := false;
end;  {exist}
```

```
procedure getbit(var b: integer);
begin
  if infilepos = 0 then
    begin
      if eof(infile) then
        done := true
      else
        begin
          blockread(infile, inbuf, maxbuf, inbytes);
          infilepos := 1;
          inbitpos := 1;
        end;
    end;
  if not done then
    begin
      b := inbuf[infilepos] and inbitpos;
      if b > 0 then b := 1;
      inbitpos := inbitpos shl 1;
      if inbitpos > 128 then
        begin
          inbitpos := 1;
          inc(infilepos);
          if infilepos > inbytes then infilepos := 0;
        end;
    end;
end;
```

```
procedure getbite(var b: integer);
begin
  if infilepos = 0 then
    begin
      if eof(infile) then
       begin
        done := true;
        b := 0
       end
      else
        begin
          blockread(infile, inbuf, maxbuf, inbytes);
          infilepos := 1;
        end;
    end;
  if not done then
    begin
      if inbitpos > 1 then
        begin
          inc(infilepos);
          writeln('Bit sync error in getbite!')
        end;
      b := inbuf[infilepos];
      inbitpos := 1;
      inc(infilepos);
```

```
    if infilepos > inbytes then infilepos := 0;
  end;
end;

procedure putbit(b: integer);
begin
 if (b > 1) or (b < 0) then writeln('Bad data passed to putbit.');
 outbite := outbite or (outbitpos * b);
 outbitpos := outbitpos shl 1;
 if outbitpos > 128 then
  begin
   inc(outfilepos);
   outbuf[outfilepos] := outbite;
   outbite := 0;
   outbitpos := 1;
   if outfilepos >= maxbuf then
    begin
     blockwrite(outfile, outbuf, maxbuf, outbytes);
     if outbytes < maxbuf then
      begin
       writeln('Disk full error.');
       done := true;
      end;
     outfilepos := 0;
    end;
  end;
```

```
end;


procedure putbite(b: integer);
begin
  if (b > 255) or (b < 0) then writeln('Bad data passed to putbite.');
  outbite := b;
  inc(outfilepos);
  outbuf[outfilepos] := outbite;
  outbite := 0;
  outbitpos := 1;
  if outfilepos >= maxbuf then
    begin
      blockwrite(outfile, outbuf, maxbuf, outbytes);
      if outbytes < maxbuf then
        begin
          writeln('Disk full error.');
          done := true;
        end;
      outfilepos := 0;
    end;
end;


procedure flushbit;
begin
  if outbitpos > 1 then
    begin
```

```
      inc(outfilepos);
      outbuf[outfilepos] := outbite;
    end;
  blockwrite(outfile, outbuf, outfilepos, outbytes);
  outbite := 0;
  outbitpos := 1;
  outfilepos := 0;
end;

function punct(ch: char): boolean;
{TRUE iff ch is not a letter or chr(255).}
  var c:  integer;
  begin
    c := ord(ch);
    if ((c > 10) and (c <= 64)) or ((c >= 91) and (c <= 96))
    or ((c >= 123) and (c <= 254)) or ((c > 0) and (c < 10)) then
      punct := true
    else
      punct := false;
  end;

function letter(ch: char): boolean;
{TRUE iff ch is a letter A-Z or a-z.}
  var c: integer;
  begin
    c := ord(ch);
```

```
  if ((c >= 65) and (c <= 90)) or ((c >= 97) and (c <= 122)) then
    letter := true
  else
    letter := false;
end;



procedure parse(var words: wd; var ch: char);
{Extracts a word or item of punctuation from input file.}
 var byt: integer;
 begin  {parse}
  words := '';
  while not (letter(ch) or punct(ch) or done) do
    begin
      getbite(byt);
      ch := chr(byt);
    end;
  if punct(ch) then
    begin
      if ord(ch) = 13 then
        begin
          words := chr(255)+chr(255);
          if not done then begin getbite(byt); ch := chr(byt) end;
        end
      else
```

```
    if ch = ' ' then
      begin
        words := ch;
        while (not done) and (ch = ' ') and (length(words) < maxstr) do
          begin
            getbite(byt);
            ch := chr(byt);
            if ch = ' ' then words := words + ch;
          end;
        if (length(words) = maxstr) and (not done) then
          begin
            getbite(byt);
            ch := chr(byt)
          end;
      end
    else
      begin
        words := ch; {return one punctuation character}
        if not done then begin getbite(byt); ch := chr(byt) end;
      end;
  end
else
  begin
    if letter(ch) then
      begin
        words := ch;
```

```
          while (not done) and letter(ch) and (length(words) < maxstr) do
            begin
              getbite(byt);
              ch := chr(byt);
              if letter(ch) then words := words + ch;
            end;
          if (length(words) = maxstr) and (not done) then
            begin
              getbite(byt);
              ch := chr(byt)
            end;
        end;
    end;
  end;  {parse}


begin
  clrscr;
  writeln('  SQUEEZE is designed to remove         This program may be copied freely ');
  writeln('  redundancy from ASCII text files,     for educational use or to try it  ');
  writeln('  making them smaller for transmission  out.  If you like it, donations   ');
  writeln('  and storage.  It also improves the    will be accepted by the author.   ');
  writeln('  security of files encrypted after     This program is believed to be    ');
  writeln('  SQUEEZING.  It uses LANGUAGE.COD,     reliable, but it is the user',ap,'s    ');
  writeln('  created with COUNT and MAKETREE.      responsibility to determine its   ');
  writeln('                                fitness for use.  No liability    ');
  writeln('  Mike Johnson                        will be assumed by the author.    ');
```

```
writeln('  P. O. Box 1151                    Copyright (C) 1988 Mike Johnson.  ');
writeln('  Longmont, CO 80502-1151              All rights reserved.            ');
if (paramcount < 3) then
  begin
    writeln;
    writeln('Syntax: SQUEEZE a|s|u|x outputfile inputfile(s)');
    writeln('a and s both mean add or squash, u and x both mean unsquash or extract.');
    writeln('Input and output files must be different.');
    writeln('Input file name(s) may include wild cards.');
    writeln('LANGUAGE.COD  (made with MAKETREE) must be in default directory.');
  end
else
  begin
    {Read in language code dictionary.}
    assign(infile,'LANGUAGE.COD');
    reset(infile,1);
    infilepos := 0;
    inbitpos := 1;
    done := eof(infile);
    newline := nil;
    last := nil;
    endfile := nil;
    if paramcount >= 1 then
      temp := paramstr(1)
    else
      temp := 'X';
```

```
temp := upcase(temp[1]);
if (temp = 'A') or (temp = 'S') then
  squish := true
else
  squish := false;
writeln('Reading in LANGUAGE.COD.');
for i := 1 to 32 do
  begin
    new(top[i]);
    top[i]^.wrd := '';
    top[i]^.count := 0;
    top[i]^.code := 0;
    top[i]^.next := nil;
    bottom[i] := top[i];
  end;
last := nil;
while not done do
  begin
    new(newbuf);
    newbuf^.wrd := '';
    getbite(strlen);
    if not done then
      begin
        for i := 1 to strlen do
          begin
            getbite(bite);
```

```
    newbuf^.wrd := newbuf^.wrd + chr(bite);
  end;
getbite(bite);
newbuf^.count := bite;
getbite(bite);
newbuf^.code := bite;
if newbuf^.count > 8 then
  begin
    getbite(bite);
    long := bite;
    newbuf^.code := newbuf^.code + (long shl 8);
    if newbuf^.count > 16 then
      begin
        getbite(bite);
        long := bite;
        newbuf^.code := newbuf^.code + (long shl 16);
        if newbuf^.count > 24 then
          begin
            getbite(bite);
            long := bite;
            newbuf^.code := newbuf^.code + (long shl 24);
          end;
      end;
  end;
newbuf^.next := nil;
if newbuf^.wrd = chr(255) then
```

```
          begin
            last := newbuf;
          end
        else
          if newbuf^.wrd = chr(255) + chr(255) then
            newline := newbuf
          else
            if newbuf^.wrd = chr(255) + chr(255) + chr(255) then
              endfile := newbuf;
        if squish then
          begin
            if ord(newbuf^.wrd[1]) > 107 then
              i := length(newbuf^.wrd) + 15
            else
              i := length(newbuf^.wrd)
          end
        else
          i := newbuf^.count;
        bottom[i]^.next := newbuf;
        bottom[i] := newbuf;
        write(#13,newbuf^.wrd,'                    ');
      end;
  end;
if newline = nil then
  writeln('Error:  newline symbol not in LANGUAGE.COD! ');
writeln;
```

```
close(infile);


{Initialize input & output files and global variables.}
outname := paramstr(2);
filefound := false;
assign(outfile,outname);
if exist(outname) then
  begin
    reset(outfile,1);
    seek(outfile,filesize(outfile))
  end
else
  rewrite(outfile,1);


outfilepos := 0;
outbitpos := 1;
outbite := 0;
if squish then start := 3 else start := 2;
for cmdparameter := start to paramcount do
  begin
    filespec := paramstr(cmdparameter);
    filemask := filespec;
    path := ”;
    repeat
      i := pos(’\’,filemask);
      if i > 0 then
```

```
    begin
      path := path + copy(filemask, 1, i);
      filemask := copy(filemask, i+1, length(filemask));
    end;
until i = 0;
findfirst(filespec, readonly + hidden + archive, s);
while doserror = 0 do
  begin
    filefound := true;
    inname := path + s.name;
    assign(infile, inname);
    reset(infile,1);
    infilepos := 0;
    inbitpos := 1;
    done := eof(infile);

    {Perform compression on input file(s).}
    if squish then
     begin
      writeln('Compressing ',inname);
      ch := chr(0);
      while not done do
        begin
          parse(words,ch);     {Find word or punctuation}
          if length(words) > 0 then
            begin
```

```
if ord(words[1]) > 107 then
  buf := top[length(words) + 15]
else
  buf := top[length(words)];
mask := 1;
while (buf^.wrd <> words) and (buf^.next <> nil) do
  buf := buf^.next;
if buf^.wrd <> words then
  begin
    highvideo;
    if words = chr(255) + chr(255) then
      writeln('Newline code not found in tree! ')
    else
      write(words);
    for i := 1 to last^.count do
      begin
        masked := last^.code and mask;
        mask := mask shl 1;
        if masked = 0 then
          putbit(0)
        else
          putbit(1);
      end;
    for i := 0 to length(words) do
      begin
        for j := 0 to 6 do
```

```
                begin
                  b := (ord(words[i]) shr j) and 1;
                  putbit(b)
                end;
              end;
            end
          else
            begin
              for i := 1 to buf^.count do
                begin
                  masked := buf^.code and mask;
                  mask := mask shl 1;
                  if masked = 0 then
                    putbit(0)
                  else
                    putbit(1);
                end;
              lowvideo;
              if words = chr(255) + chr(255) then
                writeln
              else
                write(buf^.wrd);
            end;
        end;
    end;
  mask := 1;
```

```
 for i := 1 to endfile^.count do
   begin
     masked := endfile^.code and mask;
     mask := mask shl 1;
     if masked = 0 then
       putbit(0)
     else
       putbit(1);
   end;
 end
else


 {Decompress input file.}
 begin
   writeln('Decompressing ',inname);
   cd := 0;
   mask := 1;
   cnt := 0;
   while not done do
     begin
       getbit(b);
       if b > 0 then cd := cd or mask;
       mask := mask shl 1;
       if mask <= 0 then writeln('Error: code too long.');
       inc(cnt);
       buf := top[cnt];
```

```
while (buf^.next <> nil) and
     ((cnt <> buf^.count) or (cd <> buf^.code)) do
  buf := buf^.next;
if (cnt = buf^.count) and (cd = buf^.code) then
  begin
   if buf = endfile then
    done := true
   else
    if buf^.wrd = chr(255) then
     begin
      j := 0;
      for k := 0 to 6 do
       begin
         getbit(b);
         j := j + (b shl k);
       end;
      for i := 1 to j do
       begin
         m := 0;
         for k := 0 to 6 do
          begin
            getbit(b);
            m := m + (b shl k)
          end;
         putbite(m);
         highvideo;
```

```
                write(chr(m));
              end;
            end
          else
           if buf^.wrd = chr(255) + chr(255) then
            begin
             writeln;
             putbite(13);
             putbite(10);
            end
           else
            begin
              for i := 1 to length(buf^.wrd) do
                begin
                  b := ord(buf^.wrd[i]);
                  putbite(b)
                end;
              lowvideo;
              write(buf^.wrd)
            end;
          cd := 0;
          cnt := 0;
          mask := 1;
        end;
    end;
end;
```

```
            {Finish up or loop back for more work to do.}
            flushbit;
            close(infile);
            findnext(s);
          end;
       end;

     if not filefound then writeln('No matching files found.');
     flushbit;
     close(outfile);
   end;
end.

{That's all, folks!}
```