**DeEnc - A File Encryption and Decryption Software**

Shubham Ghosh, Raju J

Department of Computer Science and Engineering

PSG Institute of Technology and Applied Research, Coimbatore, Tamil Nadu, India

**ABSTRACT**

As the amount of information sharing increases. We could possibly get a lot of discrepancies. In today’s world where information is regarded as the most valuable asset to an organisation, it is quite understandable that it attracts malicious individuals. These individuals try to seize the information and try to gain unfair advantage mostly either via extortion or replicating.This defames the organisation. Hence, it’s necessary to safeguard these kinds of information during the time it is most vulnerable, i.e., during transmission. This is where DeEnc comes into picture - to make the transmission secure by encrypting the information, so that even if it is sabotaged during transmission, it will still be virtually useless to the partakers of the sabotage.

The software is implemented in two parts viz. Encryption part and the decryption part.

Encryption: Converting the data to a form which is not understandable by anyone other than the intended user.

Decryption: Converting the non-understandable data back to understandable form.

For the software, C++ has been chosen as the language of choice. The reason being that C++ is a multi paradigm language [1]. C++ provides a myriad of general-type (templated) Abstract Data Types (ADTs) and algorithms alike, which encourages us to follow DRY principle in the process. [2][3]

**Keywords:** Encryption, Decryption, Cipher, Decipher, ADT, Automatic, Shift, Caesar, Route, Rot13, Atbash.

**1. INTRODUCTION**

**1.1 Encryption**

Data encryption translates data into another form, or code, so that only people with access to a decryption key can read it. Currently, encryption is one of the most popular and effective data security methods used by organisations. Two main types of data encryption exist - asymmetric encryption, also known as public-key encryption, and symmetric encryption. [4]

The purpose of data encryption is to protect digital data confidentiality as it is stored on computer systems and transmitted using the internet or other computer networks. These algorithms provide confidentiality and drive key security initiatives including authentication, integrity, and non-repudiation.

**1.2 Decryption**

The conversion of encrypted data into its original form is called Decryption. It decodes the encrypted information so that an authorised user can only decrypt the data because decryption requires a secret key or password. [4]



**Figure 1: Encryption-Decryption Process Overview**

**1.3 Ciphers**

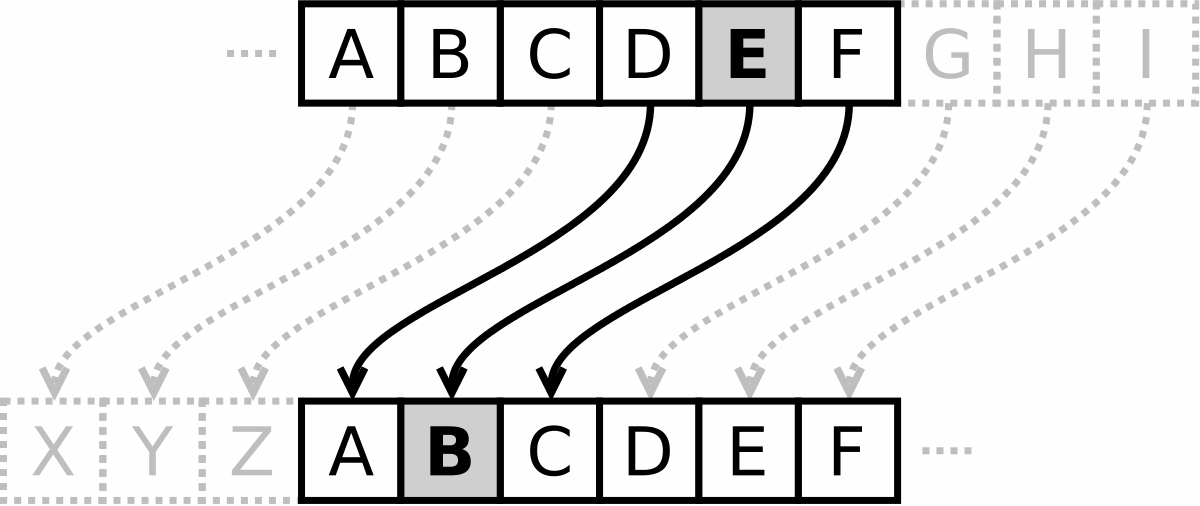
In cryptography, a cipher (or cypher) is an algorithm for performing encryption—a series of well-defined steps that can be followed as a procedure.. To encipher or encode is to convert information into cipher or code. In common parlance, "cipher" is synonymous with "code", as they are both a set of steps that encrypt a message; however, the concepts are distinct in cryptography, especially classical cryptography. The various ciphers available in DeEnc are Shift (Caesar, Rot13), Substitution (Atbash) and Transposition (Route) ciphers. Their implementation and working has been explained in detail in the upcoming sections. [5]

**1.3.1 Shift Ciphers**

Shift ciphers are concerned with “shifting” (not physically) the each character of the plaintext by some fixed integral amount [6]. This fixed amount is known as a “key”. For example, if key = 1, then the plaintext “ABCXYZ” becomes “BCDYZA”. Hence, a modulo operation is also involved here, which explains the “looping back” of ‘Z’ back to ‘A’. There are several kinds of shift ciphers, some of which have been implemented in DeEnc. Which are:

Caesar Cipher - Involves shifting the characters with 3 as the key.

Rot13 cipher - Involves shifting the characters with 13 as the key.



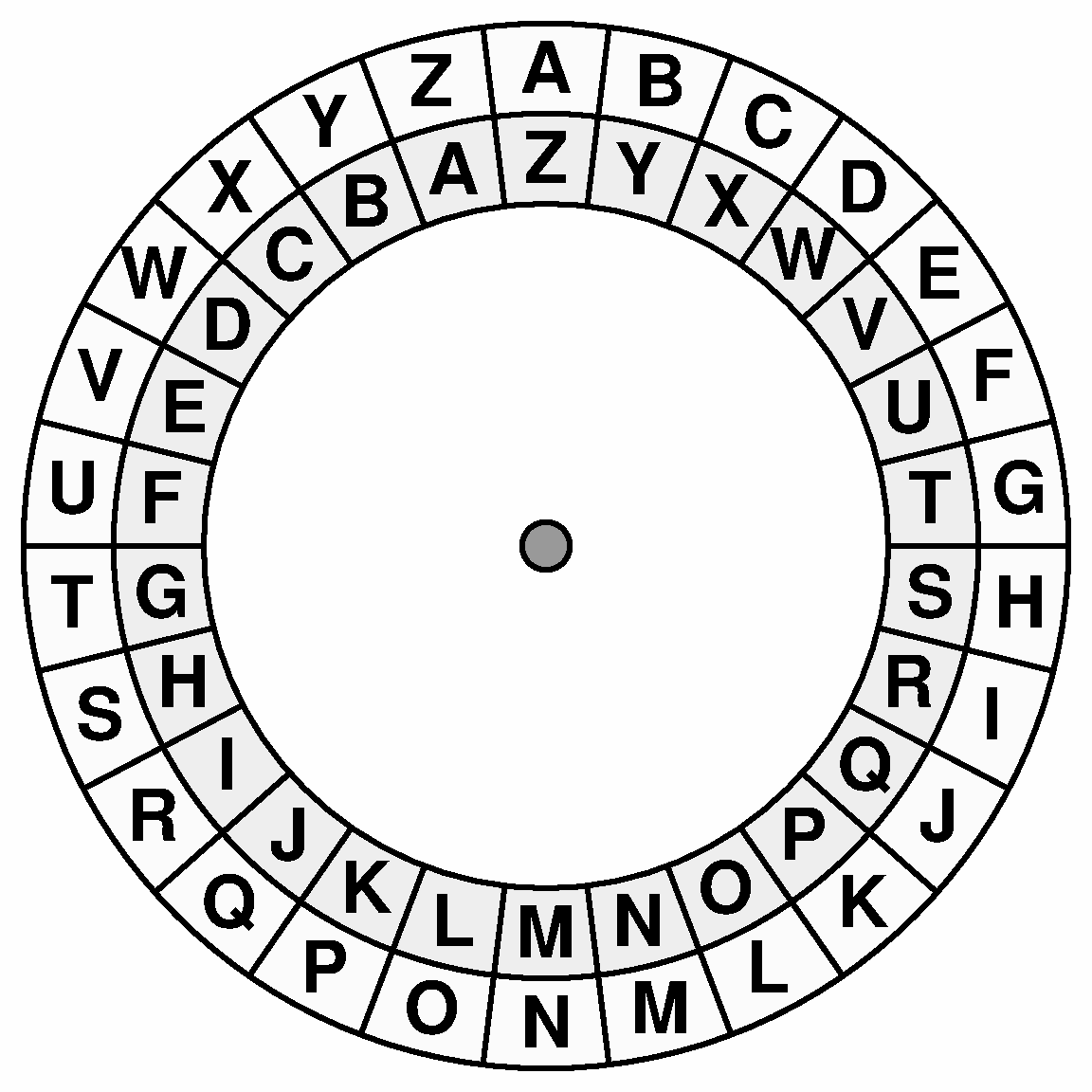
**Figure 2: An Example for Shift Cipher**

**1.3.2 Substitution Ciphers**

In substitution ciphers, every character is substituted with another character. Substitution is done using a “substitution map” which defines what character is to be replaced with what character. [7]

A substitution cipher that is implemented in DeEnc is the Atbash cipher, where in the *ith* alphabet character is replaced with the *ith* alphabet character from the end. For instance, ‘A’ is replaced with ‘Z’, ‘B’ with ‘Y’ and so on.

An example for the Atbash cipher is as follows: Let the plaintext (source) be “HELLO, WORLD!”. Upon encryption with atbash cipher, the phrase is converted to “SVOOL, DLIOW!”, which obviously looks like garbage value to the untrained eyes.



**Figure 3: Substitution Map for Atbash Cipher**

**1.3.2 Transposition Ciphers**

In Transposition ciphers, the characters of the plaintext are rearranged to form a ciphertext. No change is done to the characters themselves. [7]

An example of this cipher is Route Cipher, which has been implemented in DeEnc. It involves an integer key, say *n*. This key becomes the number of columns of a matrix, where the plaintext is to be filled row-wise. To obtain the ciphertext, the matrix is read in some order, like zig-zag or column-wise. The latter method has been implemented in DeEnc.

For instance, consider the phrase “ABORT THE MISSION” with key = 5. The matrix looks as follows:

| A | B | O | R | T |
| --- | --- | --- | --- | --- |
|  | T | H | E |  |
| M | I | S | S | I |
| O | N | \* | \* | \* |

**Figure 4: Matrix for the Given Phrase**

Hence, after reading column-wise, the ciphertext is “A MOBTINOHS\*RES\*T I\*”.

**1.4 Deciphers**

A decipher is an algorithm to decrypt the ciphertext.

However, there are a few deciphers which vary from their cipher counterparts. One example being the route cipher, which has been used in DeEnc.

**2. Literature Survey**

Cipher plays a significant role in abstracting the true nature of data; this is achieved by inducing a series of shifts and other mathematical functions. In the field of cryptography there exist several techniques for encryption/decryption and techniques can be generally classified into two major groups Conventional and Public key Cryptography. Conventional encryption is marked by its usage of a single key for both the process of encryption and decryption whereas in public key cryptography separate keys are used. Conventional techniques are further divided into Classical and Modern techniques.

Public key cryptography is also a choice when it comes to encryption but it requires excessive communication and processing resources. Our proposed methods to some extent deals with some of the drawbacks of existing techniques it includes usage of key without inducing any confusion in the primary key and changed that by generating many sub-parts of key from the primary key, similarly the key size of proposed concept may vary from 4bits or 32 bits character onwards to 64-bits ,128-bits and so on whereas on the other hand we have example of DES, AES and triple-DES, Blow-Fish that have fixed key structure.

The variation in key introduces the aspect of uncertainty which is a positive aspect when it comes to encryption, time complexity is the phenomenon that describes the effect in the output cipher text if a large text data is changed in the file. This change that occurs at the output should be sufficient in order to create a secure algorithm.

To analyse the time-consumption of the known cryptographic algorithms: triple-DES, AES and Blow-Fish and IDEA. There is one evaluating mode: different plaintexts in the same key (DPSK). As the basis of the evaluating model, the plaintext and the corresponding key are both generated by random numbers.

**2.1 Random Number Generation Model**

Generation Mechanism, evaluating one algorithm usually needs to consider time complexity and space complexity, which must be quite clear of the algorithm. However, on the premise of unknown algorithm description, theoretical analysis becomes infeasible. The evaluation model based on the random number generating mechanism can simulate the generation of plaintexts and keys that occur actually in existence, and the number of evaluating tests does not see exponential growth according to the input scale. Because if we do not use the random number, then the tests should be according to the size of data (from a few bytes to several megabytes) and the content of data to carry out. According to the principles of the statistics, the larger the number of evaluated data generated is, the more evident the statistical effect of this evaluation model is.

In this evaluation model, the plaintext consists of a packet which is written by the random number. There are two parameters for generating a packet: the pointer for the array to store the packet data and the packet size, where the array is used for storing the generating message of a packet randomly. Usually, random numbers are generated as part of the simulation, so they should closely approximate the ideal statistical properties of uniformity and independence.

A packet is associated with the simulation of a file data, so, in this evaluated system, we have defined six different evaluated packets with different sizes, in order to measure the time-consumption of each algorithm that encrypts different sizes of plaintexts. These are 4KB, 8KB, 16KB, 32KB, 64KB and 128KB. For triple-DES, because it is a block cipher, without the loss of generality, the processed plaintext is split into many blocks and a block size is 64 bits. The key length of triple-DES is 192-bit, given a plaintext P and three encryption keys K1, K2 and K3 and is defined as follows:C = EK3[DK2[EK1[P]]].

The generated plaintext does the following five functions: an initial permutation (IP); a complex function labelled fk, which involves both permutation and substitution operations and depends on a key input; a simple permutation function that switches (SW) the two halves of the data; the function fk again, and finally a permutation function that is the inverse of the initial permutation (IP1).

For AES, its block-cipher length and key length are variable, and it was limited to 128-bit block-cipher and with three cipher key strengths of it: a 128-bit, 192-bit and 256-bit encryption key, but the corresponding number of iterative rounds is 10, 12 or 14.

**2.2. The Generation of Keys**

A key is a value that causes a cryptographic algorithm to run in a specific manner and produce a specific cipher text as an output. In this evaluation model, random numbers play an important role in the use of encryption. We use the function of “rand()” to simulate the random numbers generation.

For the other two algorithms, the function of “rand()” selects the specified median of the random number to simulate the key generation. So in this evaluation system, the key length of AES can be chosen at will.

We are working with only four block symmetric algorithms like DES, Triple-DES, AES, and Blow-Fish.

Asymmetric key encryption or public key encryption is used to solve the problem of key distribution. In Asymmetric keys, two keys are used; private and public keys. Public key is used for encryption and private key is used for decryption. Because users tend to use two keys: public key, which is known to the public and private key which is known only to the user. There is no need for distributing them prior to transmission.

However, public key encryption is based on mathematical functions, computationally intensive and is not very efficient for small mobile devices. Asymmetric encryption techniques are almost 1000 times slower than Symmetric techniques, because they require more computational processing power.

**3. OBJECTIVES**

The primary objective of the software is obviously, to encrypt and decrypt the contents of a given text file or just text entered by the user.

We also aim to make the “layering” of ciphers possible, that is encrypting the already encrypted text over and over again using same/different ciphers from what was used initially.

We also aim to make the encrypted file password protected and make automatic decryption possible on it. Automatic decryption in this context stands for a mechanism to detect what ciphers were used with what key, and in which order without the intervention of the user.

Additionally, we aim to detect any tampering in the encrypted file and deny further decryption if detected.

**4. IMPLEMENTATION**

**4.1 Reading the Input**

The input (the plaintext itself) can be read either from the standard input stream or a text file, the latter is preferred as buffer overflow presents a significant threat in case of large inputs. In either case, a buffer should be used as the size of the input can vary greatly. It is to be ensured that whatever mechanism is being used, it does not tamper with the whitespaces and blank spaces, as they might play a significant role in the data’s integrity. We want the decrypted product to resemble the original text as closely as possible.

The read input is stored in a variable of appropriate data type.

**4.2 Implementing Shift Ciphers**

A general purpose module or function can be made for shift ciphers. This function shall accept the *plaintext* and *key* as the arguments. A naive approach to shift ciphers can be to insert all the alphabets (A-Z, case insensitive) into a Circular Linked List (CLL). For every plaintext[*i*], find the corresponding character in the CLL and move forward *k* number of times where *k = key*. The resultant character is ciphertext[*i*]. But this approach is very inefficient as for every character in *plaintext*, the CLL is to be searched completely to find that character. Hence, the effective time complexity comes out to be *O(m\*(n + k))*, where *m* is the number of characters in *plaintext* and *n* is the number of characters or alphabet in the source language. For example, *n =* 26 for English. An efficient search algorithm in the best possible case can bring the complexity down to *O(m\*(log(n) + k))*.

A way of optimising is to use a *hashmap* with a strong hashing algorithm to minimise collisions. The alphabets are inserted into the CLL as before, but this time a hashmap contains the location of each alphabet in the CLL, so that the searching overhead can be eliminated. Hence the hashmap contains pairs of the form <alphabet, location in the CLL>. Hence, for each plaintext[*i*], go to *hashmap*[*plaintext*[*i*]] and move *k* number of times forward. Hence, the time complexity is reduced to *O(m\*k)*. But the space complexity is now increased by a factor of *O(m/2)* on average.

The best method however, is by ASCII modification and modulo operation. In this method, the insertion and searching overhead are not only reduced, but are eliminated altogether. For each *plaintext*[*i*], *ciphertext*[*i*] = (*plaintext*[*i*] + *key*) mod 26. Hence, the effective time complexity comes out to be *O(m)*, with a space complexity of *O(1)*.

Caesar cipher, Rot13 cipher modules or functions can be implemented by simply making a call to the shift cipher function or module with *key =* 3 and *key =* 13 respectively. Hence enforcing DRY principle (Don’t Repeat Yourself) in the process.

The shift deciphers are relatively easy to implement. They are virtually the opposite of the ciphers. For instance, for *ciphertext*[*i*] we are going to move *k* places backwards instead of forwards to retrieve *plaintext*[*i*].

**4.3 Implementing Substitution Cipher**

As discussed in Section 1.3.2, substitution ciphers tend to use a substitution map. Hence a *hashmap* can be used for this purpose. The *hashmap* may contain pairs of the form <character, character>. In this particular case, an ordered map is not needed, an unordered map would suffice. Hence *ciphertext*[*i*] can be obtained using *hashmap*[*plaintext*[*i*]]. However, it does involve a space overhead in the order of *O(2m)*.

On a deeper thought, this space overhead can actually be eliminated. The hashmap can be replaced with a simple mathematical formula in case of ASCII character set. For instance, for uppercase English characters, *ciphertext*[*i*] = 155 - ASCII\_value\_of(plaintext[*i*]). Similarly, a formula for lowercase characters can also be found. Using this optimised approach, the time complexity is reduced to just *O(m)*, and the space complexity is reduced to *O(1)*. Here, *m* means the same as what it meant in section 3.2.

In this case also, the deciphers are easy to implement. We just have to follow the procedure backwards.

**4.4 Implementing Transposition Cipher**

The transposition cipher that is implemented in DeEnc in the route cipher, which has been explained elaborately in section 1.3.2. One approach is to actually construct the transposition *matrix*. It will contain *k* columns and *⌈m/k⌉* rows. Read *plaintext*[*i*] for all valid *i*. Now insert them into a temporary array (preferably a vector). If *i* is divisible by *k*, then insert the temporary vector into the *matrix* and clear the temporary vector. Once this process is over, the *matrix* is read from top to bottom, left to right. The letters are stored in that order to obtain the *ciphertext*. Hence, the time complexity comes out to be *O(m + k\*⌈m/k⌉)*, with a space complexity of *O(⌈m/k⌉ + k)*.

One of the major challenges in this cipher makes itself apparent in the decipher part. The challenge being the detection of “invalid” cells, i.e., the cells which are not part of the original plaintext. For example, the ones marked as ‘\*’ in Figure 4. These are “invalid” as the plaintext has ended before itself and they are left empty. Using a delimiter as shown is not viable, as the delimiter itself might be a part of the plaintext, which might cause ambiguity during deciphering. To tackle this, error handling features of C++ can be used. As vectors in C++ are variable sized, hence for the last three cells of the last row of the matrix in Figure 4, if implemented in C++, are actually left empty. That is, every row of a matrix in C++ can be of different sizes, sometimes referred to as a jagged matrix. This is different from languages such as C which only allow equal sized rows. Hence, performing an operation like *matrix*[3][2] will lead to an error, which can be captured. This error can mean that the cell is an “invalid” cell. This information can be then used to decipher accordingly. Obviously, all of these operations have to be done with a few control checks such as the row counter should never exceed *length\_of\_ciphertext/key*. Violation of such constraints can prove to be dangerous as they might cause segmentation faults.

**4.5 Implementing Automatic Decryption**

**4.5.1 Keeping Track of the Ciphers used**

The track of ciphers used can be kept using any linear data structure, such as arrays, vectors etc. during the encryption phase. But the best data structure in this regard is undoubtedly the stack. A unique “code” can be generated for every cipher which shall also contain the key involved in the cipher. The code generated has to look as if it is a random string of numbers, hence “salting” mechanisms become a necessity [8].

For instance, a cipher with *key* = 10 may have a code CX18310 and the same cipher with *key = 20* might have a code CX18320. But these can be cracked easily if the sample space is large enough. Hence after “salting”, something like H2SH182DCX183103DNF29 looks much more secure, even though the actual code is still present. Only a fixed length random sequence is added before and after the actual code. As the code before salting is always the same and so is the code after salting, the actual code can be “unsalted” easily while deciphering. The “salted” code can then be encrypted with a cipher whose key will be known by the software at both the sender and the receiver sides. These codes can be pushed onto the stack in order. The stack contents are then written to the file in a specific format so that they can be used at the receiver side during automatic decryption.

**4.5.2 Making the file password protected**

For making the file password protected, two approaches can be followed. The first one being encrypting the password with a cipher whose key will be known at both the sender and receiver sides, and then adding the password to the file. The receiver can then extract the password, decrypt it and verify whether the entered password is the same as the one present in the file. However, this approach does pose the challenge of the password being variable in length, which can cause difficulty during extraction of the password as the receiver does not know what the size of the password is, and hence does not know where to start extracting from and where to end. This can however be managed by making the use of new line feeds. Such as keeping the password alone in a specific line number from the bottom and so on.

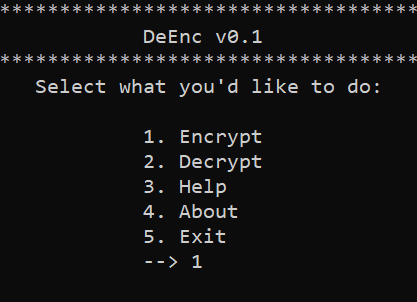
The above challenges can be controlled further via the use of hashing, which maps a sequence of characters to another sequence of characters which is fixed in length, irrespective of the length of the input [9]. It is to be noted however that while encryption is a reversible process, hashing is not. In this approach, once the hashed password is extracted from the file at the receiver side, the password entered by the user is hashed and then both are compared. Many hashing algorithms exist in the market today such as SHA128, SHA256, SHA3 128 and so on [10].

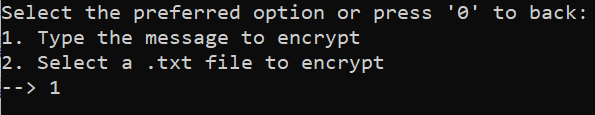
**4.5.3 Ensuring the Integrity of the File**

Since all the information added to the file such as the ciphers used, password etc. are in a specific format which is known by the software at both the sender and receiver sides, any deviation from that format shall mean that integrity is lost. This however, is a very crude way of ensuring integrity of the file and can be improved using checksums and hashes.

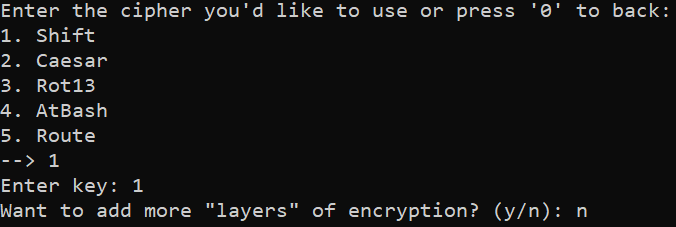
**5. RESULTS AND ANALYSIS**

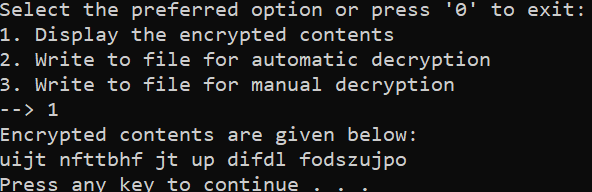
Encryption-

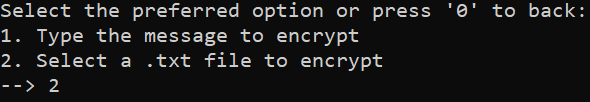




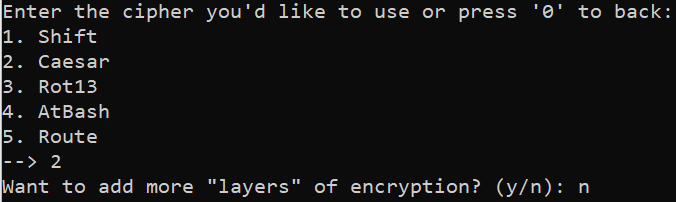


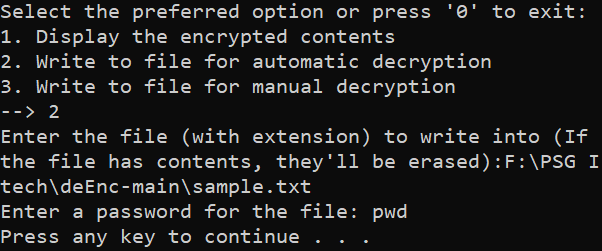


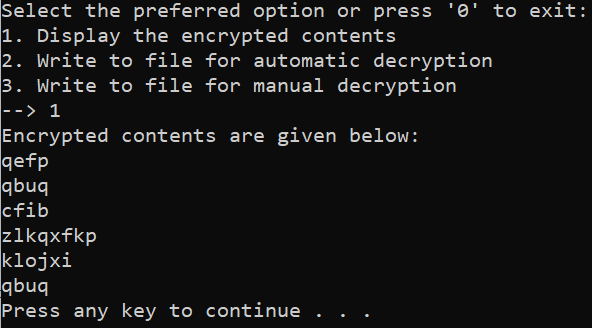






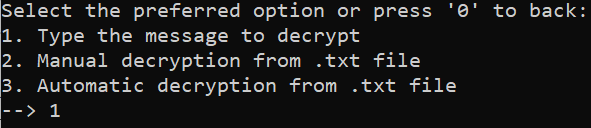




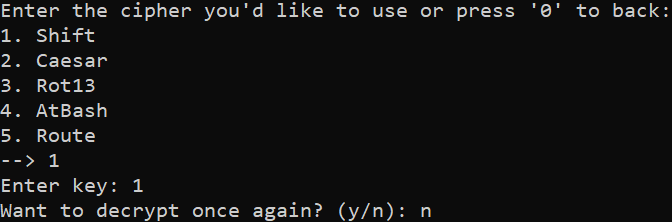


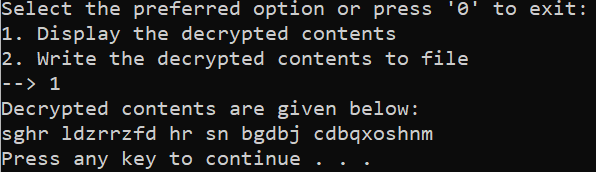
Decryption-

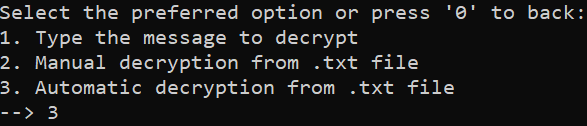




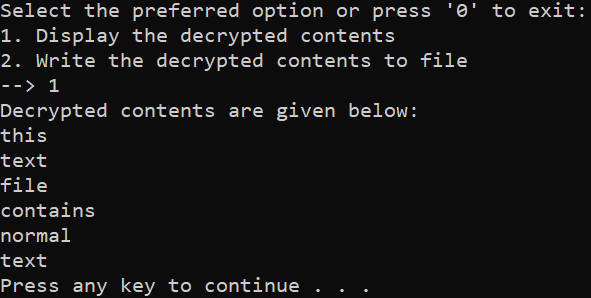












The results of this study after testing the application that implements the shift ciphers, substitution cypher and transposition cipher is shown above. The application provides functionalities like encryption, decryption, help, about etc. Here encryption and decryption is allowed for simple inputs as well as for text files with .txt extension.

Encryption functionality gives the option to view the encrypted content and also to write the encrypted data or text file to another text file either for manual decryption or automatic decryption.

Encryption of data or text files is allowed with multiple ciphers viz Shift, Caesar, Rot13, Atbash, Route etc. The application also provides multiple layers of encryption with the above listed ciphers. It gives three options: to view the encrypted content; to write the encrypted text to a file for automatic/manual decryption.

To write to a file for automatic decryption, the application asks the user to set a password, which is required in the decryption process.

Decryption allows us to decrypt a data or text file with the five ciphers as listed above. For normal data decryption it gives the option to view the decrypted content. And for text file decryption if manual encryption was selected while performing encryption then in the decryption process, the application will ask for the ciphers to be used to decrypt. If automatic decryption was selected then, the application will only ask for the password set at the time of encryption to write for automatic decryption. That makes the file back to its original form.

This shows the application is highly secured as a result of including multiple layers of encryption via multiple ciphers being used.

The application was executed multiple times and it shows consistent results, the results are evident as the screenshots of the output’s above shows. The application works fine as intended.

Since this is the very first version of DeEnc, errors are expected which can be rectified and the application is open for improvements. And since this application is open users are allowed to report the bugs which is useful to do enhancements to the application and to yield better results.

**6. CONCLUSION AND FUTURE ENHANCEMENTS**

The review has helped shed light on certain aspects of information security, such as integrity. It has also shed some light on the automatic encryption and decryption along with password protection of a text file. Though not a fool-proof strategy, it does provide a basic idea upon which newer and better technologies implementing the same can be built.

There are scopes for improvement in the existing software, which are explained next. Firstly, the number and variety of ciphers is quite low which can be easily fixed by adding new ciphers to the system. Secondly, the system does not quite address methods of encrypting numeric values, which can also be fixed via the addition of newer ciphers. The system also lacks a proper User Interface, which can be added via other external modules, though not secure.

**7.OBSERVATIONS:**

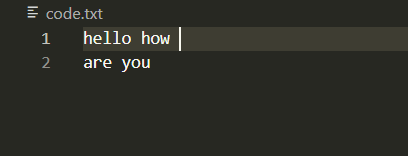
The observations of encryption and decryption were recorded in a table and following results were obtained:

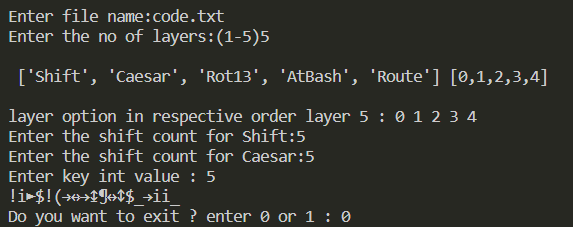
1. There are two kinds of input allowed for encryption and decryption:

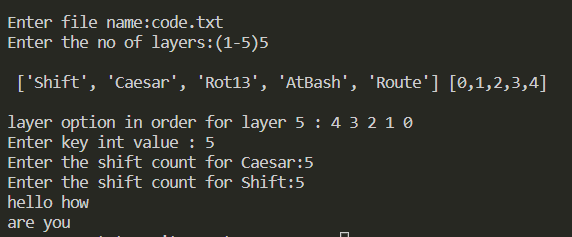
i) input string

ii) text file input

1. The application was implemented using five different algorithms namely shift, caesar, atbash, rot13, route etc.
2. The encryption and decryption happens on a variety of strings including alphabets(both uppercase and lowercase), integer numbers, special characters.
3. Encryption was observed for varying length of input string and decryption of the encrypted string yielded original string and also time taken for each encryption and decryption was noted in the table.
4. The application provides layered encryption and decryption.
5. Layered encryption and decryption allows to incorporate different combinations of cipher algorithms in order to increase the security of messages.
6. Encrypted string while decryption using all the cipher algorithms that were used during encryption yielded the same results.
7. Layered encryption had a constraint of passing the reverse order of algorithms chosen during encryption.
8. Decryption did not yield the original string for a different order of the same algorithm chosen during encryption.
9. Time taken for the encryption and decryption of text files took a larger time compared to input strings.
10. The application provided enhanced security of the encrypted message by enabling a password while writing to a file.
11. Another constraint of the application is that it only allows encryption and decryption for ASCII characters.
12. Encryption and decryption were limited to only five algorithms.
13. Encryption and decryption were limited to only text files.
14. The application showed greater consistency with accurate matching between decrypted text and the original text.







**7. REFERENCES**

1. https://msritse2012.wordpress.com/2013/01/31/the-concept-of-multi-paradigm-programming-language-manjunath-m/
2. Barbara Liskov and Stephen Zilles. 1974. Programming with abstract data types. SIGPLAN Not. 9, 4 (April 1974), 50–59.
3. https://en.wikipedia.org/wiki/Don't\_repeat\_yourself
4. Dwi Liestyowati 2020 J. Phys.: Conf. Ser. 1477 052062
5. Mohammed, Abdalbasit & Varol, Nurhayat. (2019). A Review Paper on Cryptography. 1-6. 10.1109/ISDFS.2019.8757514.
6. Singh, Rohit & Kumar, Naveen. (2018). A Review Paper on Cryptography of Modified Caesar Cipher.
7. Poonia, Preeti & Kantha, Praveen. (2016). Comparative Study of Various Substitution and Transposition Encryption Techniques. International Journal of Computer Applications. 145. 24-27. 10.5120/ijca2016910783.
8. Gauravaram, Praveen. (2012). Security Analysis of salt||password Hashes. Proceedings - 2012 International Conference on Advanced Computer Science Applications and Technologies, ACSAT 2012. 25-30. 10.1109/ACSAT.2012.49.
9. Sobti, Rajeev & Ganesan, Geetha. (2012). Cryptographic Hash Functions: A Review. International Journal of Computer Science Issues, ISSN (Online): 1694-0814. Vol 9. 461 - 479.
10. Sahu, Aradhana & Ghosh, Samarendra. (2017). Review Paper on Secure Hash Algorithm With Its Variants. 10.13140/RG.2.2.13855.05289.