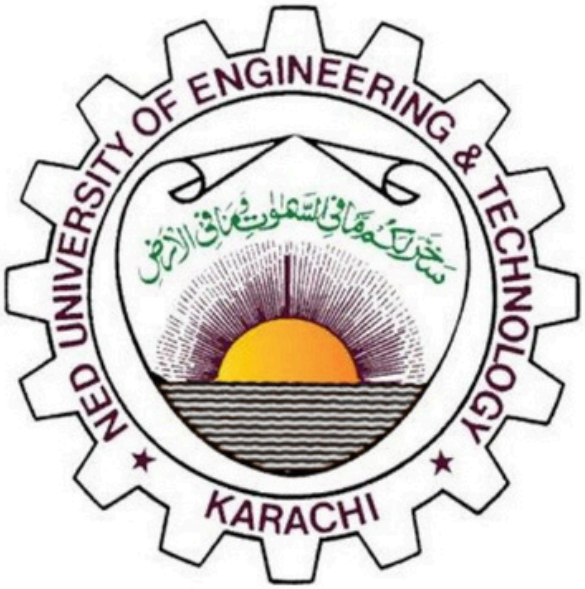
CCP - Project report

PF CT-175



By:

Laiba - CT-25060

Areeba - CT-25070

Minahil - CT-25066

Teacher: Sir Abdullah

REPORT

CAESAR CIPHER

CCP - PROJECT REPORT

PROGRAMMING FUNDAMENTALS

TABLE OF CONTENTS

INTRODUCTION

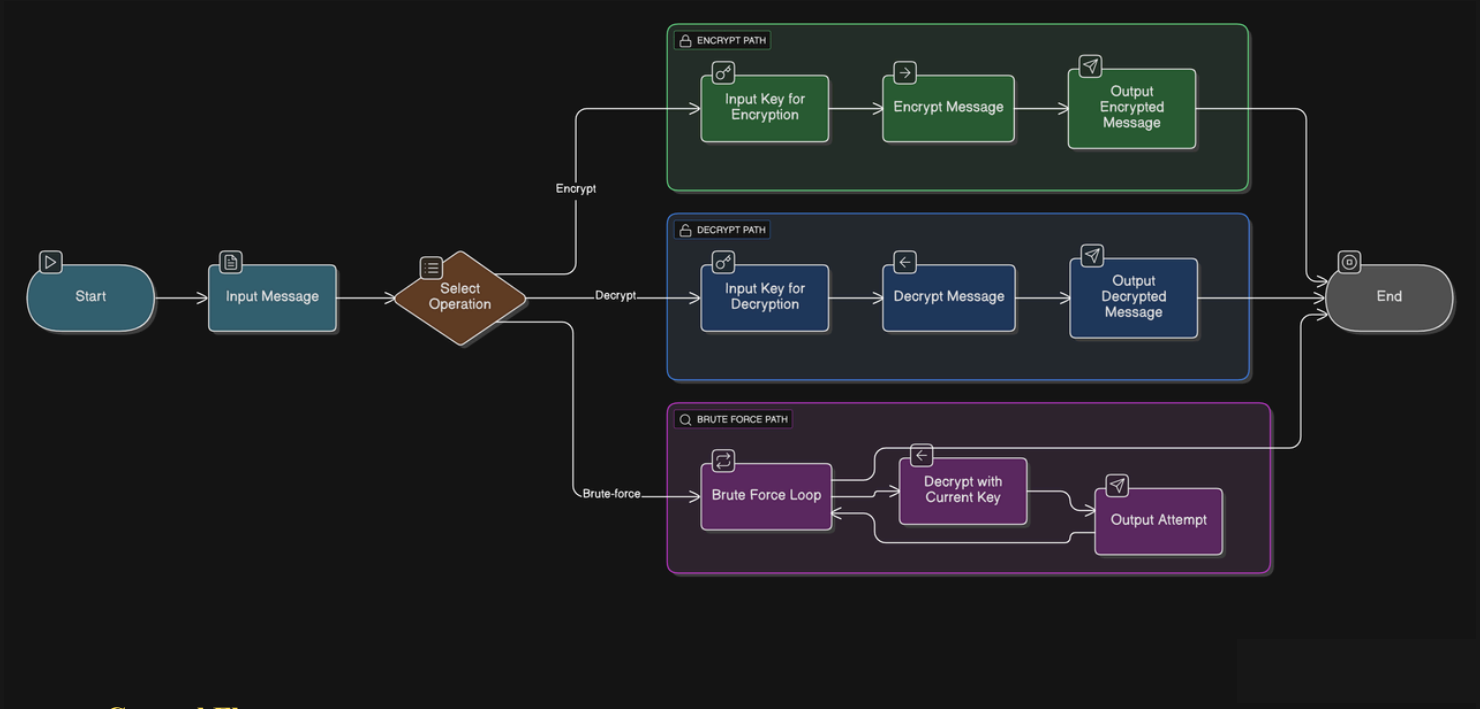
This project, presents a complete implementation of the Caesar Cipher algorithm in the C programming language. The Caesar Cipher is one of the oldest and simplest substitution cipher techniques, used historically for secure communication. It operates by shifting each letter in a plaintext message by a fixed number of positions down the alphabet.

Our program provides a user-friendly console application that demonstrates three core functionalities of the cipher:

1. **Encryption:** Transforming a plaintext message into ciphertext using a user-specified shift key.
2. **Decryption:** Recovering the original plaintext from ciphertext using the correct key.
3. **Brute Force Technique**: Automatically generating all 25 possible decryptions, showcasing a fundamental cryptanalysis technique used when the key is unknown.

This report details the algorithm's logic, our implementation approach, and the program's operational results, highlighting both the utility and the inherent security weaknesses of the Caesar Cipher.

FLOWCHART AND STRUCTURE



Control Flow:

1. Initialization: The program starts and presents a formatted menu to the user with three options: Encryption, Decryption, and Brute Force.
2. User Input: The user's integer choice is captured via a switch statement.
3. Function Routing:
   - Case 1 (Encryption): Prompts the user for a message and a secret key, then passes them to the encrypt() function.
   - Case 2 (Decryption): Prompts the user for an encrypted message and the correct key, then passes them to the decrypt() function.
   - Case 3 (Brute Force): Prompts the user for an encrypted message and automatically calls the bruteforce() function to display all decryption possibilities.

1. Execution & Output: The respective function processes the input and prints the result directly to the console before the program terminates.

Algorithmic Core

Algorithmic Core:

The security transformation is handled by a shared algorithm within the encrypt and decrypt functions:

- Character Processing: The algorithm iterates through each character of the input string.

- Case Preservation: It checks if a character is an uppercase (A-Z) or lowercase (a-z) letter. Non-alphabet characters are ignored and preserved.

- Mathematical Transformation:
  - Encryption Formula: new_status=(status+key)%26;
  - Decryption Formula: new_status=(status - key + 26) % 26;
  - The + 26 ensures the result is non-negative before the modulo operation, which handles wrap-around at the ends of the alphabet.

The bruteforce function leverages the decryption logic in a loop, testing every possible key from 1 to 25 to demonstrate the cipher's vulnerability to exhaustive search attacks.

IMPLEMENTATION

This section details the practical execution and code-level decisions made during development.

1. Core Program Logic

- **User Interface & Control Flow:** The main() function uses a switch-case statement to direct execution to three distinct operational modes based on user input, creating a clear, menu-driven console application.

- **Input Handling:** The program uses fgets() for safe string input, preventing buffer overflow. The line text[strcspn(text, "\n")] = 0; efficiently removes the trailing newline character. while(getchar()!='\n'); is used to clear the input buffer after scanf(), ensuring subsequent fgets() calls work correctly.

2. Function Specifications

- encrypt() **&** decrypt()**:** Both functions perform in-place modification of the input character array. They share identical control structures but use the inverse mathematical operation (+ key for encryption, - key for decryption).

- bruteforce()**:** This function demonstrates a complete cryptanalytic attack. It iterates through all 25 non-trivial keys (1-25) and for each key, it operates on a temporary copy (temp_copy) of the original message to avoid altering the original input during the analysis.

3. Technical Details

- **Character Encoding:** The logic leverages ASCII arithmetic. Subtracting 'A' or 'a' converts a character to its 0-based index. Adding this index back to 'A' or 'a' converts the result back to a character, ensuring case preservation.

- **Robustness:** The implementation correctly handles wrap-around using the modulo operator and preserves non-alphabetic characters, making it suitable for processing standard sentences with spaces and punctuation.

RESULT AND OUTPUT

This section demonstrates the successful execution of the program's three core functionalities with verified input-output examples.

1**. Encryption**

- Input: "Hello World!" with key 3
- Output: "Khoor Zruog!"
- Verification: 'H' → 'K', 'e' → 'h' confirms correct forward shift while preserving case and ignoring non-alphabet characters.

Figure 1: Encryption of "Hello World!" with shift key 3

2. Decryption

Input: "Khoor Zruog!" with key 3

Output: "Hello World!"

Verification: Successfully recovers the original plaintext, demonstrating the reversibility of the cipher.

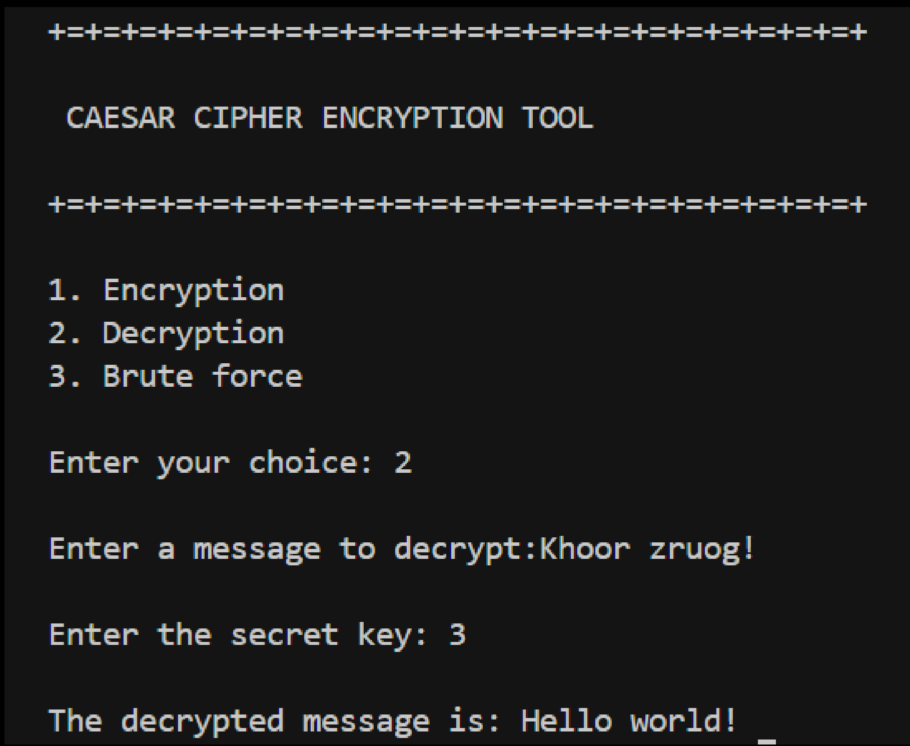


Figure 2: Decryption of "Khoor Zruog!" with shift key 3

3. **Brute Force Attack**

Input: "Wklv lv d whvw."

Output: All 25 possible decryptions (key 1-25)

Verification: The correct plaintext "This is a test." is visibly present at key 3, demonstrating the cipher's vulnerability to exhaustive search attacks.

```
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

 CAESAR CIPHER ENCRYPTION TOOL

+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

1. Encryption
2. Decryption
3. Brute force

Enter your choice: 3

Enter a message to generate all possible encryptions for:Wklv lv d whvw
Vjku ku c vguv
Uijt jt b uftu
This is a test
Sghr hr z sdrs
Rfgq gq y rcqr
Qefp fp x qbpq
Pdeo eo w paop
Ocdn dn v ozno
Nbcm cm u nymn
Mabl bl t mxlm
Lzak ak s lwkl
Kyzj zj r kvjk
Jxyi yi q juij
Iwxh xh p ithi
Hvwg wg o hsgh
Guvf vf n grfg
Ftue ue m fqef
Estd td l epde
Drsc sc k docd
```

Figure 3: Brute-force decryption output for "Wklv lv d whvw."

CHALLENGES AND SOLUTIONS

1. Brute Force Display Logic

- Challenge: Initially, the brute-force function displayed incorrect key mappings (e.g., plaintext appearing at Key 2 instead of Key 3).
- Solution: Discovered inconsistent decryption formulas between the decrypt() and bruteforce() functions. Standardized both to use (status - key + 26) % 26.

2**. Input Buffer Management**

- Challenge: Using scanf() for menus then fgets() for text input caused skipped inputs, as scanf() leaves newlines in the buffer.
- Solution: Added while(getchar() != '\n'); to clear the input buffer after each scanf() call.

3**. Case Sensitivity in Alphabet Processing**

- Challenge: The initial implementation struggled with mixed-case messages, sometimes converting uppercase to lowercase or ignoring case entirely.
- Solution: Implemented separate handling for uppercase ('A'-'Z') and lowercase ('a'-'z') character ranges using different base characters in the shift calculation.

4. Non-Alphabetic Character Preservation

- Challenge: Early versions of the program would corrupt spaces, punctuation, and numbers during encryption/decryption.
- Solution: Added conditional checks to only transform alphabetical characters, leaving all other characters unchanged in the output.

CODE

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
void encrypt(char message[], int key); //encryption function declaration
void bruteforce(char message[]);//bruteforce function declaration
void decrypt(char message[], int key); //decryption function declaration

int main() {
    char text[500]="";
    int s_key;
    int part;
    printf("+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=\n\n");
    printf(" CAESAR CIPHER ENCRYPTION TOOL\n\n");
    printf("+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=\n\n");
    printf("1. Encryption\n2. Decryption\n3. Brute force\nEnter your choice: ");
    scanf("%d", &part);
    while(getchar()!='\n');

    switch(part){
        case(1):{
    printf("\nEnter a message to encrypt:");
    fgets(text,500,stdin);
    text[strcspn(text, "\n")]=0;
    printf("\nEnter the secret key: ");
    scanf("%d", &s_key);
    encrypt(text, s_key);
    break;
    }
case(2): {
 printf("\nEnter a message to decrypt:");
 fgets(text,500,stdin);
 text[strcspn(text, "\n")]=0;
 printf("\nEnter the secret key: ");
 scanf("%d", &s_key);
 decrypt(text, s_key);
 break;
 }
```

```
        case(3):{
    printf("\nEnter a message to generate all possible decryptions for:");
    fgets(text,500,stdin);
    text[strcspn(text, "\n")]=0;
    bruteforce(text);
    break;
        }
         default: {
    printf("Invalid selection");
    }
    }
    }
    void decrypt(char message[], int key){
    int length=strlen(message);
    int status, new_status;
    int i;
    for(i=0;i<length;i++){
    char ch=message[i];
    if(ch>='A' && ch<='Z'){
     status=ch-'A';
     new_status=(status - key + 26) % 26;
    message[i]=new_status+'A';
    }
else if(ch>='a' && ch<='z'){
 status=ch-'a';
 new_status=(status+key)%26;
 message[i]=new_status+'a';
 }
 }
 printf("The encrypted message is: %s", message);
 }
void encrypt(char message[], int key){
 int length=strlen(message);
 int status, new_status;
 int i;
 for(i=0;i<length;i++){
 char ch=message[i];
 if(ch>='A' && ch<='Z'){
 status=ch-'A';
 new_status=(status+key)%26;
 message[i]=new_status+'A';
 }
```

```
    else if(ch>='a' && ch<='z'){
    status=ch-'a';
    new_status=(status+key)%26;
    message[i]=new_status+'a';
    }
    }
    printf("The encrypted message is: %s", message);
    }
  void bruteforce(char message[]){
  int length=strlen(message);
  int status, new_status;
  int i;
  char temp_copy[500];
  for(int j=1;j<=25;j++){
  strcpy(temp_copy, message);
  for(i=0;i<length;i++){
  char ch=temp_copy[i];
  if(ch>='A' && ch<='Z'){
  status=ch-'A';
  new_status=(status-j+26)%26;
  temp_copy[i]=new_status+'A';
  }
  else if(ch>='a' && ch<='z'){
  status=ch-'a';
  new_status=(status-j+26)%26;
  temp_copy[i]=new_status+'a';
  }
  }
 printf("%s", temp_copy);
  printf("\n");
  }
  }
```

INSIGHTS

Working on this project gave us a practical understanding of both the fundamentals and the limitations of cryptographic systems. While implementing the Caesar Cipher, it became clear why it's now considered a teaching tool rather than a secure method. The most glaring weakness is its extremely limited keyspace—with only 25 possible keys, a brute-force attack can break the cipher in seconds, as our own program demonstrates.

This experience highlighted a critical lesson in cybersecurity: complexity does not equal security. A system can be logically sound (and the Caesar Cipher's algorithm is) yet still be completely insecure due to a simple lack of possible combinations. It made us appreciate the immense importance of a large keyspace in modern encryption, like that used in AES, where the number of possible keys is astronomically large.

CONCLUSION

This project successfully demonstrated the implementation and analysis of the Caesar Cipher, one of the oldest known cryptographic techniques. Through hands-on coding, we developed a functional tool capable of encryption, decryption, and brute-force attacks, which provided practical insight into fundamental cryptography concepts.

The project highlighted both the historical significance and the severe security limitations of the Caesar Cipher. Its small key space and vulnerability to exhaustive search attacks make it completely unsuitable for modern security needs. This understanding naturally leads to an appreciation for modern encryption standards like AES, which use complex mathematical transformations and enormously large key spaces that even a computer takes years to decode, since currently it is impossible.

In essence, this project served as a bridge between historical cryptography and modern security principles, emphasizing why encryption has evolved from simple substitution to highly complex algorithms.