ITRI 615 – COMPUTER SECURITY
PROJECT DOCUMENTATION

MNISI G – 34292748
MAKATISE OB – 31807240

Date of submission: 19 May 2023

Contents

1 Introduction **1**
1.1 Vigenère Cipher 1
1.1.1 Implementation of the Vigenère Cipher 1
1.2 Transposition Cipher 5
1.2.1 Implemetation of the Transposition Cipher 5
1.3 Vernam Cipher 7
1.3.1 Implementation of the Vernam Cipher 8
1.4 B&G InHouse 10
1.4.1 Implementation of the B&G InHouse 11

2 User Manual **17**
2.1 Start Up 17
2.2 Text Encryption 17
2.2.1 B&G InHouse 18
2.3 File Encryption 19
2.3.1 B&G InHouse 20

3 Platforms Used **22**
3.1 Visual Studio 2022 22
3.2 GitHub 22

4 Bibliography **23**

List of Figures

1 Start Up Page 17
2 Text Encryption Page 18
3 B&G Text Encryption Page 19
4 File Encryption Page 20
5 B&G File Encryption Page 21

List of Tables

1 ASCII Table 11

Listings

1 Vigenère Cipher Text Source Code 2
2 Vigenère Cipher File Source Code 4
3 Transposition Cipher Text Source Code 5
4 Transposition Cipher File Source Code 6
5 Vernam Cipher Text Source Code 8
6 Vernam Cipher File Source Code 9
7 B&G Cipher Text Source Code 12
8 B&G Cipher File Source Code 14

Section 1

1 Introduction

The modern definition of information security involves the protection of information assets, and components that facilitate the storage, usage, and transmission of the said information – emphasis being on the protection of information to preserve its confidentiality, integrity, and availability (Singh, 2003). According to Pfleeger *et al.,* (2015:74), encryption is the process of encoding a message so that its contents are concealed, and decryption is the reverse of such a process – the decoding of the message to understand its' meaning.

This project evaluates 4 types of cryptography methods (Vigenère, Transposition, Vernam, and Custom ciphers) and how each method is implemented to achieve encryption of the messages and files.

1.1 Vigenère Cipher

The Vigenère cipher, the first of its kind since the invention of the Caesar cipher, was created in the middle of the sixteenth century. It was created by Blaise de Vigenère, a former diplomat turned scholar, and published in 1856.

1.1.1 Implementation of the Vigenère Cipher

The Vigenère uses a simple polyalphabetic substitution method, where the plaintext is encrypted using a unique key, and the letters are shifted in terms of the key.

Messages

Encryption Takes in two parameters, plaintext, and key, and returns a cipher text. Then uses a Dictionary to store all of the alphabet indexes from A to Z, A being 0 and Z being 25. If the character is not a letter, the algorithm appends the cipher text as it is. If the character is a letter, the algorithm calculates the index of the letter in plaintext and the cipher key using the 'AlphabetOrder Dictionary', which arranges the letters in alphabetical order based on their position. It uses the FirstOrDefault method, which obtains the first element of the resulting sequence, or the default value if no element is found. The algorithm calculates the sum of two indexes obtained from the plaintext and the cipher key and then takes the modulus of 26. This index is then used to retrieve the corresponding letter from the Alphabet Order dictionary. The final index of the plaintext is retrieved in alphabetical order. The processed characters in the plaintext are then used to generate the ciphertext.

Decryption The decryption process follows the same steps as the encryption process, with one exception. Instead of calculating the sum of two indexes obtained from the plaintext and the cipher key, the algorithm calculates their

difference. The modulus of 26 is then applied to ensure that the resulting index is within the valid range and returns the plaintext.

Listing 1: Vigenère Cipher Text Source Code

```
class VigenereText
{
    public String VigenereTextEncrypt(String plain, String key)
    {
        Dictionary<sbyte, char> AlphabetOrder = new Dictionary <
            sbyte, char> ();

        AlphabetOrder.Add(0, 'A');
        ......................
        AlphabetOrder.Add(25, 'Z');

        key = key.ToUpper();
        plain = plain.ToUpper();

        string encryptedText = "";

        int i = 0;
        try {
            foreach (char elements in plain)
            {
                if (!Char.IsLetter(elements))
                {
                    encryptedText += elements;
                }
                else
                {
                    sbyte firstOrder = AlphabetOrder.
                        FirstOrDefault(x => x.Value == elements)
                        .Key;
                    sbyte lastOrder = AlphabetOrder.
                        FirstOrDefault(x => x.Value == key[i]).
                        Key;
                    sbyte finalValue = (sbyte)(firstOrder +
                        lastOrder);

                    if (finalValue > 25)
                    {
                        finalValue -= 26;
                    }
                    encryptedText += AlphabetOrder[finalValue];
                    i++;
                }
                if (i == key.Length)
                {
                    i = 0;
                }
            }
            return encryptedText;
        } catch (Exception e)
        {
            return ("Error" + e.Message);
        }
```

```
47          }
48
49          public String VigenereTextDecrypt(String mixed, String key)
50          {
51              Dictionary <sbyte, char> AlphabetOrder = new Dictionary
                    <sbyte, char>();
52
53              AlphabetOrder.Add(0, 'A');
54              ......................
55              AlphabetOrder.Add(25, 'Z');
56
57              key = key.ToUpper();
58              mixed = mixed.ToUpper();
59              string decryptedText = "";
60
61              int i = 0;
62
63              try
64              {
65                  foreach (char elements in mixed)
66                  {
67                      if (!Char.IsLetter(elements))
68                      {
69                          decryptedText += elements;
70                      }
71                      else
72                      {
73                          sbyte firstOrder = AlphabetOrder.
                                FirstOrDefault(x => x.Value == elements)
                                .Key;
74                          sbyte lastOrder = AlphabetOrder.
                                FirstOrDefault(x => x.Value == key[i]).
                                Key;
75                          sbyte finalValue = (sbyte)(firstOrder -
                                lastOrder);
76                          if (finalValue < 0)
77                          {
78                              finalValue += 26;
79                          }
80                          decryptedText += AlphabetOrder[finalValue];
81                          i++;
82                      }
83                      if (i == key.Length)
84                      {
85                          i = 0;
86                      }
87                  }
88                  return decryptedText;
89              }
90              catch (Exception e)
91              {
92                  return ("Error" + e.Message);
93              }
94          }
95      }
```

Files

The Vigenère can encrypt files as strings or bytes, in this case bytes. In our implementation, the key is usually generated so that it can be used with ease in the decryption method. When it encrypts, it converts the contents into non-alphabetic ones. If it is a text file, it converts each character to the corresponding letter; A becomes 0, and so on. The extension of the file is usually changed to "vig" to show that it is encrypted, and it uses the same method to decrypt and uses the generated key.

Listing 2: Vigenère Cipher File Source Code

```
1         public void VigenereFileDecrypt(string encryptedFiles,
              string keyFile, string decryptedFile)
2         {
3             byte[] encryptedBytes;
4
5             using (FileStream fileStream = new FileStream(
                  encryptedFiles, FileMode.Open))
6             {
7                 encryptedBytes = new byte[fileStream.Length];
8                 fileStream.Read(encryptedBytes, 0, encryptedBytes.
                      Length);
9             }
10
11            byte[] bytesOfKey;
12            using (FileStream fileStream = new FileStream(keyFile,
                  FileMode.Open))
13            {
14                bytesOfKey = new byte[fileStream.Length];
15                fileStream.Read(bytesOfKey, 0, bytesOfKey.Length);
16            }
17
18            byte[] decryptedData = new byte[encryptedBytes.Length];
19            VigenereEncrypt(encryptedBytes, bytesOfKey, ref
                  decryptedData);
20            if (encryptedFiles.Contains(".vig"))
21            {
22                encryptedFiles.Replace(".vig", "");
23                using (FileStream fileStream = new FileStream(
                      decryptedFile, FileMode.Create))
24                {
25                    fileStream.Write(decryptedData, 0, decryptedData
                          .Length);
26                }
27            }
28            else
29            {
30                using (FileStream fileStream = new FileStream(
                      decryptedFile, FileMode.Create))
31                {
32                    fileStream.Write(decryptedData, 0, decryptedData
                          .Length);
33                }
34            }
35        }
```

1.2 Transposition Cipher

A transposition cipher is a type that scrambles the order of letters. One of the oldest ways to scramble words in this form was created in ancient Greece by the Egyptians and Greeks.

1.2.1 Implemetation of the Transposition Cipher

Messages

It uses an abstract class to implement security algorithms, including an alphabet dictionary, and has a constructor that initializes the object alphabet dictionary. The algorithm has two abstract classes, encrypt and decrypt, which are used in this project. The algorithm inherits from the CreateDictionary class, with a constructor that takes the key and determines the number of rows in the transposition matrix.

It uses two major functions that will be discussed further which are called:

- The Transpose() method takes in two parameters. The value of the encryption key determines the number of rows and based on the length of the message and the key, it calculates the number of rows and columns in the transposition matrix. This method then calls a private method called FillArray, which creates a 2-dimensional char array representing the transposition matrix.
- The FillArray() method fills the remaining positions with asterisks (*) if there are no characters left in the plain text to fill the positions in the columns. Both methods use the same functions which are the Process() which uses the FillArray.

Listing 3: Transposition Cipher Text Source Code

```
1  class TranspositionText : CreateDictionary
2      {
3          private string TransposeText(string message, Mode mode)
4          {
5              int rows = encryptionKey;
6              int columns = (int)Math.Ceiling((double)message.Length /
                    (double)rows);
7              char[,] matrix = FillArray(message, rows, columns, mode)
                   ;
8              string finalText = "";
9              foreach (char c in matrix)
10             {
11                 finalText += c;
12             }
13             return finalText;
14         }
15
16         private char[,] FillArray(string message, int numberOfRows,
               int numberOfColumns, Mode mode) //
17         {
18             int charPositionInArr = 0;
```

```
19              int lengthOfArr = 0, widthOfArr = 0;
20              char[,] matrix = new char[numberOfRows, numberOfRows];
21
22              switch (mode)
23              {
24                  case Mode.Encrypt:
25                      lengthOfArr = numberOfRows;
26                      widthOfArr = numberOfRows;
27                      break;
28                  case Mode.Decrypt: //If the mode is decrypt, the
                        length is the number of columns and the width is
                         the number of rows
29                      matrix = new char[numberOfRows, numberOfRows];
30                      widthOfArr = numberOfRows;
31                      lengthOfArr = numberOfRows;
32                      break;
33              }
34
35              for (int i = 0; i < widthOfArr; i++)
36              {
37                  for (int j = 0; j < lengthOfArr; j++)
38                  {
39                      if (charPositionInArr < message.Length)
40                      {
41                          matrix[j, i] = message[charPositionInArr];
42                      }
43                      else
44                      {
45                          matrix[j, i] = '*';
46                      }
47
48                      charPositionInArr++;
49                  }
50              }
51              return matrix;
52          }
53          internal enum Mode
54          {
55              Encrypt, Decrypt
56          }
57      }
```

Files

It uses the same key to encrypt and decrypt. It uses a method called Transpose that takes in three parameters, which include the input, output, and status of the encryption method. The status is the one that will be used in the buttons to determine whether it has to encrypt or decrypt . The transpose method is the one that converts the content of the file into bytes. A key is usually generated after encrypting the file, it can be used to decrypt the file.

Listing 4: Transposition Cipher File Source Code

```
public void Transpose(string input, string output, int status)
        {
            byte[] bytes;

            using(FileStream fileStream = new FileStream(input,
                FileMode.Open))
            {
                bytes = new byte[fileStream.Length];
                fileStream.Read(bytes, 0, bytes.Length);
            }

            byte[] inputArr = bytes;
            Array.Reverse(inputArr, 0, inputArr.Length);

            if (status == 1)
            {
                using(FileStream fileStream = new FileStream(output
                    + ".trans", FileMode.Create)) {
                    fileStream.Write(inputArr, 0, inputArr.Length);
                }
            } else
            {
                if(input.Contains(".trans"))
                {
                    input.Replace(".trans", " ");
                    using (FileStream fileStream = new FileStream(
                        output, FileMode.Create))
                    {
                        fileStream.Write(inputArr, 0, inputArr.
                            Length);
                    }
                } else
                {
                    using (FileStream fileStream = new FileStream(
                        output,FileMode.Create))
                    {
                        fileStream.Write(inputArr, 0, inputArr.
                            Length);
                    }
                }
            }

        }
```

1.3 Vernam Cipher

The Vernam cipher was invented by Gilbert Sandford Vernam (1890-1960) in 1918. The Vernam Cipher is based on randomization. The plaintext character from a message is mixed with one character stream from a key stream. If the keystream was indeed random, the resulting ciphertext will also be random with no relation to the plaintext. To create the ciphertext, the logical operator (exclusive-or) XOR is applied to the individual bits of the plaintext and the key

stream.

1.3.1 Implementation of the Vernam Cipher

Messages

The encryption algorithm uses XOR (exclusive OR) to encrypt and decrypt. It first stores the encrypted or decrypted text and iterates through the plain text. It uses the XOR operations in the plain text, the character, and the cipher text will be stored in a variable which will be the one that is encrypted.

It takes the remainder of the cipher text and calculates the index of the key character for the plain text and divides it by the length of the key which ensures that the key is used in both operations (encryption and decryption). The key character will be retrieved from the plaintext and the index of the key, and this key will be used in the XOR operation. The XOR operator then compares each bit of the two operands and produces a result.

Listing 5: Vernam Cipher Text Source Code

```
namespace Cryptography_Project.Vernam
{
    internal class VernamText
    {
        public string VernamEncryptionAndDecryption(string plainText
            , string key)
        {
            var encryptedText = new StringBuilder();

            for (int cipherText = 0; cipherText < plainText.Length;
                cipherText++)
            {
                char character = plainText[cipherText];

                uint charCode = (uint)character;

                int keyPosition = cipherText % key.Length;

                //take the key character at the calculated index
                char keyChar = key[keyPosition];

                uint keyCode = (uint)keyChar;

                uint combinedCode = charCode ^ keyCode;

                char combinedChar = (char)combinedCode;

                encryptedText.Append(combinedChar);
            }
            return encryptedText.ToString();
        }
    }
}
```

Files

Encryption The Vernam algorithm takes in 3 parameters which are input, output, and key. There is an array of bytes that will be utilized to read the original file. The FileStream class is used to read the input file into the byte array. When it reads the file, it also generates a random key with the exact length of the original file, the key is containing random bytes. Then it writes the key bytes into the key. It creates a duplicate array of the original byte array and is passed to the encryption method. Then bytes are written into the file.

Decryption It works the same way as the Encryption, just that when it decrypts it takes in the key that was generated by the encryption method to decrypt the file.

Method The encryption method takes in three parameters, inputArray, keyArray, and outputArray. It checks whether all those parameters have the same length, then start with the XOR to create a new array.

Listing 6: Vernam Cipher File Source Code

```
internal class VernamFiles
    {
        public void VernamFileEncrypt(string inputFile, string
            encryptedFile, string keyFile)
        {
            byte[] originalBytes;

            byte[] keyBytes = new byte[originalBytes.Length];
            Random rand = new Random();
            rand.NextBytes(keyBytes);

            byte[] encryptedBytes = new byte[originalBytes.Length];
            VernamEncrypt(originalBytes, keyBytes, ref
                encryptedBytes);
        }

        private void VernamEncrypt(byte[] inputBytes, byte[]
            keyBytes, ref byte[] outBytes)
        {
            if (inputBytes.Length != keyBytes.Length || (keyBytes.
                Length != outBytes.Length))
            {
                MessageBox.Show("The length of the original, key,
                    and cipher must be the same.", "Error",
                    MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
            else
            {
                for (int i = 0; i < inputBytes.Length; i++)
                {
                    outBytes[i] = (byte)(inputBytes[i] ^ keyBytes[i
                        ]);
                }
            }
        }
```

```
29
30          public void VernamFileDecrypt(string encryptedFiles, string
                keyFile, string decryptedFile)
31          {
32              byte[] encryptedBytes;
33
34              using (FileStream fileStream = new FileStream(
                    encryptedFiles, FileMode.Open))
35              {
36                  encryptedBytes = new byte[fileStream.Length];
37                  fileStream.Read(encryptedBytes, 0, encryptedBytes.
                        Length);
38              }
39
40              byte[] keyFileBytes;
41              using (FileStream fileStream = new FileStream(keyFile,
                    FileMode.Open))
42              {
43                  keyFileBytes = new byte[fileStream.Length];
44                  fileStream.Read(keyFileBytes, 0, keyFileBytes.Length
                        );
45              }
46
47              byte[] decryptedData = new byte[encryptedBytes.Length];
48              VernamEncrypt(encryptedBytes, keyFileBytes, ref
                    decryptedData);
49
50              if (encryptedFiles.Contains(".ver"))
51              {
52                  encryptedFiles.Replace(".ver", "");
53                  using (FileStream fileStream = new FileStream(
                        decryptedFile, FileMode.Create))
54                  {
55                      fileStream.Write(decryptedData, 0, decryptedData
                            .Length);
56                  }
57              }
58              else
59              {
60                  using (FileStream fileStream = new FileStream(
                        decryptedFile, FileMode.Create))
61                  {
62                      fileStream.Write(decryptedData, 0, decryptedData
                            .Length);
63                  }
64              }
65          }
66      }
```

1.4 B&G InHouse

Our in-house method, called B&G InHouse, is a custom solution that we developed ourselves. The method employs an ASCII Table to create a custom cipher, which involves three steps: swapping characters, reversing them, and shifting them using the ASCII Table

0	**NUL**	16	**DLE**	32	**SP**	48	**0**	64	**@**	80	**P**	96	**‘**	112	**p**
1	**SOH**	17	**DC1**	33	**!**	49	**1**	65	**A**	81	**Q**	97	**a**	113	**q**
2	**STX**	18	**DC2**	34	**”**	50	**2**	66	**B**	82	**R**	98	**b**	114	**r**
3	**ETX**	19	**DC3**	35	**#**	51	**3**	67	**C**	83	**S**	99	**c**	115	**s**
4	**EOT**	20	**DC4**	36	**$**	52	**4**	68	**D**	84	**T**	100	**d**	116	**t**
5	**ENQ**	21	**NAK**	37	**%**	53	**5**	69	**E**	85	**U**	101	**e**	117	**u**
6	**ACK**	22	**SYN**	38	**&**	54	**6**	70	**F**	86	**V**	102	**f**	118	**w**
7	**BEL**	23	**ETB**	39	**’**	55	**7**	71	**G**	87	**W**	103	**g**	119	**v**
8	**BS**	24	**CAN**	40	**(**	56	**8**	72	**H**	88	**X**	104	**h**	120	**z**
9	**HT**	25	**EM**	41	**)**	57	**9**	73	**I**	89	**Y**	105	**i**	121	**y**
10	**LF**	26	**SUB**	42	*	58	**:**	74	**J**	90	**Z**	106	**j**	122	**z**
11	**VT**	27	**ESC**	43	+	59	**;**	75	**K**	91	**[**	107	**k**	123	**{**
12	**FF**	28	**FS**	44	**,**	60	<	76	**L**	92	\	108	**l**	124	—
13	**CR**	29	**GS**	45	**-**	61	=	77	**M**	93	]	109	**m**	125	**}**
14	**SOH**	30	**RS**	46	**.**	62	>	78	**N**	94	ˆ	110	**n**	126	∼
15	**SI**	31	**US**	47	/	63	**?**	79	**O**	95	_	111	**o**	127	**DEL**

Table 1: ASCII Table

1.4.1 Implementation of the B&G InHouse

Messages

Our custom code is very easy to implement but it is very hard to crack although it does not have any key. It uses two major functions which are swapping and reversing.

The swapping method takes in a string and swaps the first and the last letter of the string and takes out the middle letters of the string and inserts it back between the last and first letter to form a new string.

The second function which is the reverse text takes the whole message and uses a for loop to iterate through it and reverses the whole string and returns it. The custom algorithm implements both of these methods by taking in the plain text as an input and outputting the cipher text. It converts the input text into a character array and iterates each character in the array.

It performs an operation of moving each value character in the ASCII table forward by three positions, which is done by adding 3 when encrypting and removing three when decrypting. The new string will be the encrypted text.

Example: it can encrypt ”custom algorithm” to ”hmynwtlqf%rtyxzr”

- Step 1: Swapping - custom algorithm = mustom algorithc
- Step 2: Reversing - mustom algorithc = chtirogla mostum
- Step 3:ASCII Table - chtirogla mostum = hmynwtlqf%rtyxzr

Listing 7: B&G Cipher Text Source Code

```
namespace Cryptography_Project.Custom
{
    internal class CustomText
    {
        public String swapText(String plainText)
        {
            int indexFirstPosition, indexLastPosition, temp;
            string newPlainText, middlePosition, firstPosition,
                lastPosition;

            indexFirstPosition = 0;                     //index of
                first position
            indexLastPosition = plainText.Length - 1;   //index of
                last position

            middlePosition = plainText.Substring(1,
                indexLastPosition - 1); //middle position

            //swapping the position of the first and last character
            temp = indexFirstPosition;
            indexFirstPosition = indexLastPosition;
            indexLastPosition = temp;

            //getting the first and last character
            firstPosition = plainText.Substring(indexFirstPosition,
                1);
            lastPosition = plainText.Substring(indexLastPosition, 1)
                ;

            //new string
            newPlainText = firstPosition + middlePosition +
                lastPosition;

            return newPlainText;
        }

        public String reverseText(String inputText)
        {
            string reversedText = "";
            for (int i = inputText.Length - 1; i >= 0; i--) //for
                loop to reverse the string
            {
                reversedText += inputText[i];
            }
            return reversedText;
        }

        public String encryptedText(string plainText)
        {
            char[] charArray = plainText.ToCharArray();

            string newCipherText = "";
            char charInArray, newCharInArray;
            int ASCIIValue, newASCIIValue;

            for (int i = 0; i < plainText.Length; i++) //for loop to
```

```
                move each character 5 positions forward in ASCII
                table
49              {
50                  charInArray = charArray[i];
51
52                  ASCIIValue = (char)charInArray;
53                  newASCIIValue = ASCIIValue + 5; //moving 5 positions
                        forward in ASCII table
54
55                  newCharInArray = (char) newASCIIValue;
56                  newCipherText += newCharInArray;
57              }
58              return newCipherText;
59          }
60
61          public String decryptedText(string plainText)
62          {
63              char[] charArray = plainText.ToCharArray();
64
65              string newCipherText = "";
66              char charInArray, newCharInArray;
67              int ASCIIValue, newASCIIValue;
68
69              for (int i = 0; i < plainText.Length; i++)
70              {
71                  charInArray = charArray[i];
72
73                  ASCIIValue = (char)charInArray;
74                  newASCIIValue = ASCIIValue - 5; //moving 5 positions
                        backward in ASCII table
75
76                  newCharInArray = (char) newASCIIValue;
77                  newCipherText += newCharInArray;
78              }
79              return newCipherText;
80          }
81      }
82  }
```

Files

This code uses a substitution cipher, which replaces one value with a different byte value based on the key generated from the password/key. It starts with a method called InitializeTable() for creating a substitution table for the substituting algorithm. The code initializes a byte array with byte values ranging from 0 to 255 and creates a two-dimensional byte array to hold the substitution table. The table array is then populated with unique byte values that are derived from the values in the first array, based on its position in the table. The generated substitution table will be used to encrypt and decrypt data using the substitution cipher algorithm

The file is read in as a byte array, and a key equal to the length of the file is created by duplicating the password up to that point. The algorithm then uses MD5 Cryptography and UTF8 Encoding to encrypt and decrypt data streams

based on the ASCII values of the plaintext characters. The ASCII values of all characters in the input file are transformed and kept in a table array. The system gets a key from the user and converts it into bytes, which have the same length as the plaintext.

A for loop is employed to shift the content of the original file by 256 values of a byte, and the plaintext is replaced with the corresponding ciphertext. Then, using an ASCII table it replaces each byte in the file with a different byte value based on the key. With the same file extension as the original file, the encrypted file is saved to the user's preferred location. To ensure that the encryption was effective, a message box finally appears. To ensure security, the code uses a hashing algorithm called MD5 to convert the user's password into a fixed, shorter key value. This key is utilized to encrypt the file, and a matching MD5 hash of the password is required to decrypt it back to its original plaintext. Since the encryption and decryption use the same key, this is considered a symmetric cryptographic algorithm

This algorithm is effective since it uses MD5 with UTF8 encoding for authentication.

- MD5 is a cryptographic hash function and secure hash algorithm that can identify certain data corruption issues. It was primarily developed for the secure encryption of data during transmission and the authentication of digital certificates.
- UTF-8 is an encoding system designed for Unicode that enables the translation of any Unicode character to a unique binary string and vice versa.

Listing 8: B&G Cipher File Source Code

```
public partial class CustomFileForm : Form
{
    byte[] abcd;
    byte[,] byteTable;

    public CustomFileForm()
    {
        InitializeComponent();
    }

    private void InitializeTable()
    {
        abcd = new byte[256];
        for (int i = 0; i < 256; i++)
        {
            abcd[i] = Convert.ToByte(i);
        }

        byteTable = new byte[256, 256];
        for (int i = 0; i < 256; i++)
        {
            for (int j = 0; j < 256; j++)
            {
                byteTable[i, j] = abcd[(i + j) % 256];
```

```
25                 }
26             }
27         }
28
29         private void Textbtn_Click(object sender, EventArgs e)
30         {
31             try
32             {
33                 byte[] fileContents = File.ReadAllBytes(plainTextbox
                       .Text);
34                 byte[] passwordTemp = Encoding.ASCII.GetBytes(
                       passwordTextbox.Text);
35                 byte[] keys = new byte[fileContents.Length];
36                 byte[] encrypted = new byte[fileContents.Length];
37
38                 for (int i = 0; i < fileContents.Length; i++)
39                 {
40                     keys[i] = passwordTemp[i % passwordTemp.Length];
41                 }
42
43                 if (encryptRadiobtn.Checked)
44                 {
45                     for (int i = 0; i < fileContents.Length; i++)
46                     {
47                         byte value = fileContents[i];
48                         byte cipherKey = keys[i];
49                         int valueIndexes = -1, keyIndexes = -1;
50
51                         for (int j = 0; j < 256; j++)
52                         {
53                             if (abcd[j] == value)
54                             {
55                                 valueIndexes = j;
56                                 break;
57                             }
58                         }
59                         for (int j = 0; j < 256; j++)
60                         {
61                             if (abcd[j] == cipherKey)
62                             {
63                                 keyIndexes = j;
64                                 break;
65                             }
66                         }
67                         encrypted[i] = byteTable[keyIndexes,
                               valueIndexes];
68                     }
69                 }
70
71                 if (decryptRadiobtn.Checked)
72                 {
73                     for (int i = 0; i < fileContents.Length; i++)
74                     {
75                         byte value = fileContents[i];
76                         byte cipherKey = keys[i];
77                         int valueIndexes = -1, keyIndexes = -1;
78
```

```
79                          for (int j = 0; j < 256; j++)
80                          {
81                              if (abcd[j] == cipherKey)
82                              {
83                                  keyIndexes = j;
84                                  break;
85                              }
86                          }
87                          for (int j = 0; j < 256; j++)
88                          {
89                              if (byteTable[keyIndexes, j] == value)
90                              {
91                                  valueIndexes = j;
92                                  break;
93                              }
94                          }
95                          encrypted[i] = abcd[valueIndexes];
96                      }
97                  }
98              }
99              catch
100             {
101                 MessageBox.Show("File is in use");
102             }
103         }
104     }
```

Section 2

2 User Manual

The application can be obtained either a USB or by downloading it from GitHub. [1] The application can run on any Windows laptop it does not need any requirements. When the application is obtained, the user can click the setup.exe file and set up the application and follow the instructions below.

2.1 Start Up

When launching the application, you will be directed to the Start-Up page, which features a numbered layout of its components in the figure below, accompanied by a brief description for each corresponding number:

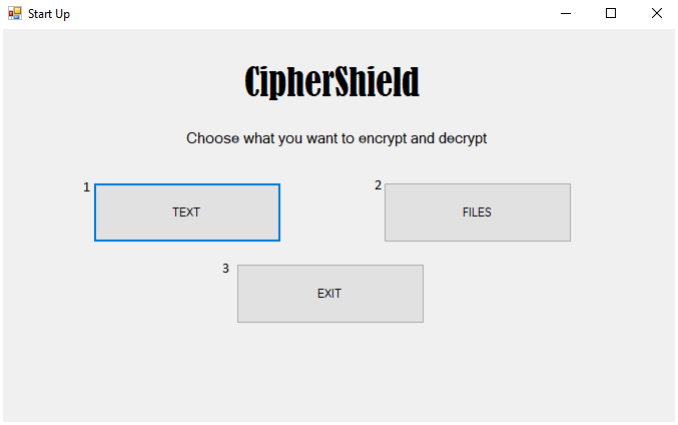

Figure 1: Start Up Page

1. This button will let you go to the Text Encryption Algorithms where you will encrypt and decrypt text.
2. This button will direct you to the file encryption and decryption operations.
3. Once you have finished using the application, simply click on this button to exit.

2.2 Text Encryption

After clicking the Text you will be redirected to this Text Page which also has its own components:

[1] : https://visualstudio.microsoft.com/vs//

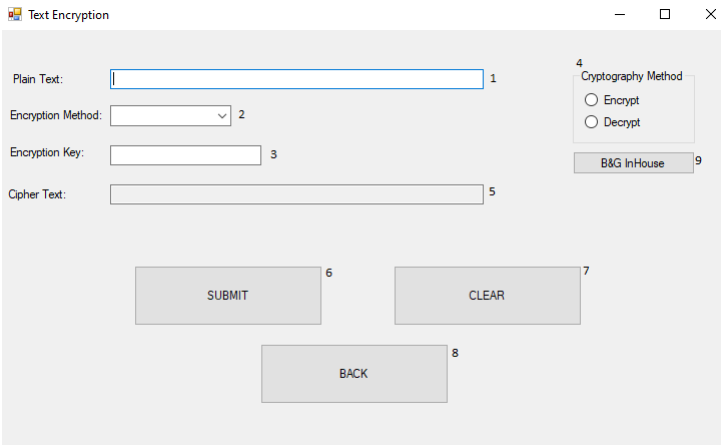

Figure 2: Text Encryption Page

1. Textbox where you input the text you want to encrypt.
2. Drop-down list for the different kinds of algorithms.
3. Enter the key (note: keys differ with each algorithm).
4. Radio buttons for encrypting or decrypting.
5. After choosing numbers 1 - 4, press the submit button
6. Output of the encrypted text.
7. Clearing button to clear the textboxes, dropdown, and radio buttons.
8. Back button to go back to the Start Up page.
9. Going to the InHouse algorithm since it uses different operations than the other algorithms.

Instructions
With regards to the text encryption methods: the user can input any message they want to encrypt and any key. **The Transposition and Vernam ciphers** only takes numbers as keys, and The **Vigenère cipher** takes letters only as encryption key. After putting in the key, you can choose any encryption method and choose to encrypt. When you want to decrypt make sure that the key is the same, if not it will not decrypt.

2.2.1 B&G InHouse

After clicking the B&G InHouse you will be redirected to this page which also has its own components:

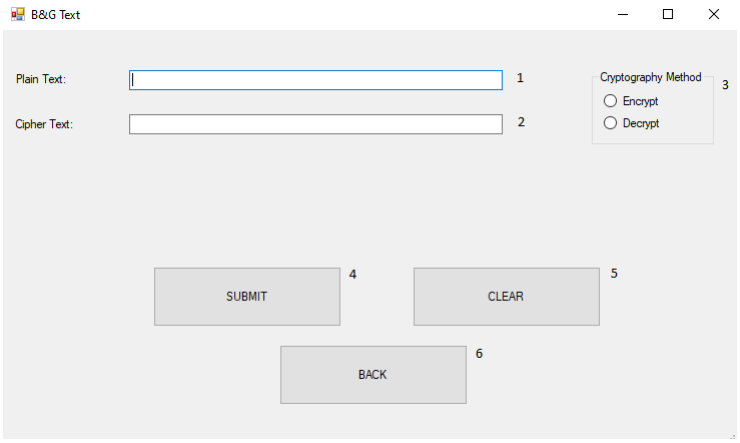

Figure 3: B&G Text Encryption Page

1. Textbox where you input the text you want to encrypt.
2. Output of the encrypted text.
3. Radio buttons for encrypting or decrypting.
4. After choosing numbers 1 - 3, press the submit button
5. Clearing button to clear the textboxes and radio buttons.
6. Back button to go back to the Text Encryption page.

Instructions
With B&G InHouse, the user can simply enter the message they want to encrypt and choose a method. No key is needed.

2.3 File Encryption

After clicking the File button you will be redirected to this Text Page which also has its own components:

1. Path of the file
2. Button to locate the file you want to encrypt or decrypt.
3. Path of the key file
4. Button to locate the key file
5. Path of the output file
6. Button to save the file

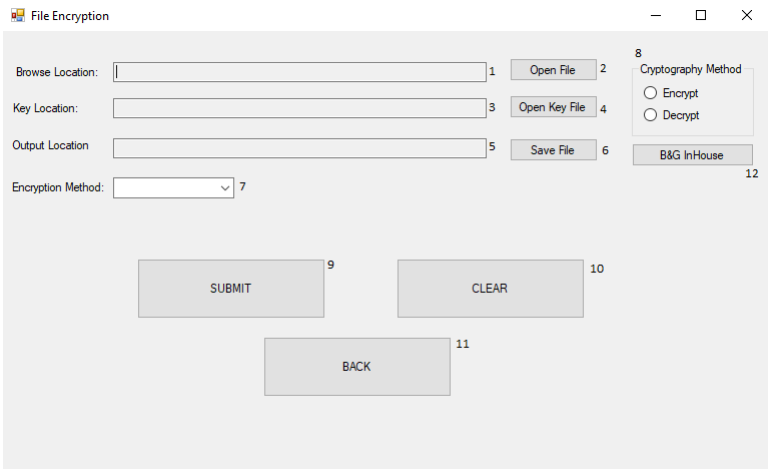

Figure 4: File Encryption Page

7. Drop-down list for the different encryption methods
8. Radio buttons for encrypting or decrypting.
9. After you are done, press the submit button
10. Clearing button to clear the textboxes and radio buttons.
11. Back button to go back to the File Encryption page.
12. Button to go to the B&G InHouse File Encryption.

Instructions
With regards to the file encryption methods: the user can press the open file and choose any file they want to encrypt, the path of the file will be displayed, and the user will then choose a key file and save the file that they want to encrypt. After that, they can choose any algorithm they want to use and choose to encrypt. When it is done a message will pop up. When you want to decrypt you select the decrypted file, in **Vernam cipher** you will select the key that is outputted and save the file, and press decrypt to get it to the original state. As well as the other algorithms.

2.3.1 B&G InHouse

After clicking the B&G InHouse you will be redirected to this page which also has its own components that will be discussed shortly.

To encrypt a file, you first select any file on your machine (such as docx, pdf, png, jpg, or any other file type), which will be displayed in the file path. Next, you enter a random key and choose to encrypt the file. Once you're done, press

Submit and save the new file with any name you want. To decrypt the file, you follow the same steps but select the encrypted file instead.

1. Path of the file
2. Button to locate the file you want to encrypt or decrypt.
3. The user does not need to input any key
4. The user will generate any random key, the key for encrypting should be the same for encrypting
5. Radio buttons for encrypting or decrypting.
6. After you are done, press the submit button
7. Clearing button to clear the textboxes and radio buttons.
8. Back button to go back to the File Encryption page.

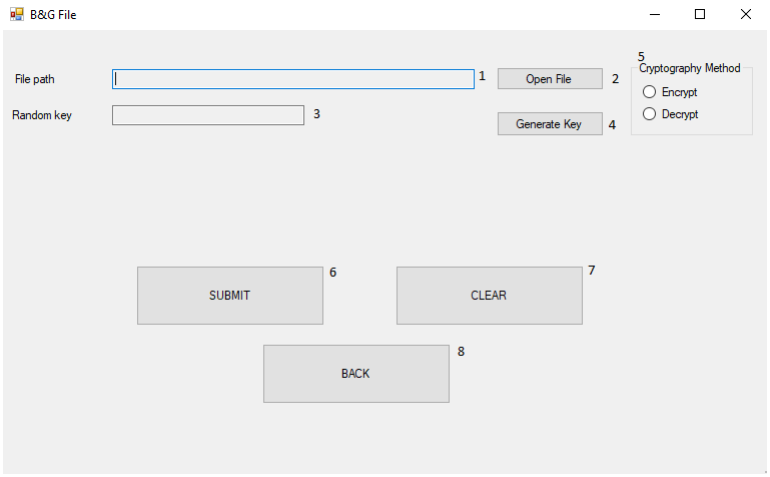

Figure 5: B&G File Encryption Page

Instructions
When you click Open File, you will go to any directory in your local machine, you select any file and click Open. The directory of the file will be shown in your File path on the application. Then you generate a random key, that will be used to encrypt and decrypt. It is a symmetrical key. You then click the Encrypt radio button and then Submit. It will lead you to the local directory where you have to name the new file. After naming it you will press save and the file will be encrypted. With encryption, you follow the same steps but you select the encrypted file and you select the decrypt radio button.

Section 3

3 Platforms Used

In the development of our project, we have chosen to utilize both Visual Studio 2022 [2] and GitHub. [3]

3.1 Visual Studio 2022

For building, editing, and GUI (Graphical User Interfaces), as well as deploying applications, Visual Studio is the ideal platform to use. We utilized the .NET Framework in C# to develop our application. We used it because of the following reasons (Microsoft, 2023):

1. Cross-platform compatibility: C# is easy to develop applications in all Operating systems. Which makes it more versatile.
2. Object-oriented programming: C# is well suited for creating complex applications which support inheritance, abstract classes, and more
3. Large community: In C# there are a lot of resources that can assist you with troubleshooting.

3.2 GitHub

In our project, we utilized GitHub throughout every development process. It makes version control easy and collaborates with others easily. We used it because of the following reasons (Kharlantseva, 2023):

1. Version control: makes it easy to track every process and allows for easy rollback to a stable version when changes have been made that break the code
2. Integration with other apps: GitHub can easily integrate with other IDEs such as Visual Studio 2022 and many more. Making it easy to automate the development process

[2] : https://visualstudio.microsoft.com/vs//
[3] `https://github.com/givenmnisi6/Cryptography-Project`

4 Bibliography

Kharlantseva, M. 2023, *What Is GitHub? Everything You Need to Know* [Blog post].
`https://everhour.com/blog/what-is-github/` Date of access: 16 May 2023.

Microsoft. 2023. *Visual Studio Subscriptions and Benefits.*
`https://visualstudio.microsoft.com/subscriptions/` Date of access: 16 May 2023.

Pfleeger, C.P., Pfleeger, S.L. & Margulies, J. 2015. *Security in Computing (5th Edition).* Prentice Hall Press.

Singh, S. 2003. The History of Cryptography: How the history of code-breaking can be used in the mathematics classroom with resources on a new CD-ROM. *Mathematics in School*:2-6.