Hill Cipher: Known-Plaintext Attack and Ciphertext-Only Attack

Sandra Matthies
FB03
Hochschule Niederrhein University of Applied Sciences
Krefeld, Germany

***Abstract*—This project deals with the Hill Cipher vunerability to calculate the key from plaintext ciphertext pairs. The Known-Plaintext and Ciphertext-Only Attack implementation has two methods for calculating the key. The key calculation can be achieved with the adjugate matrix or with the eigenvectors and eigenvalues of a matrix. Depending on which calculation method is selected, the attacks are carried out using the selected method. The adjugate matrix is used to calculate a key if all conditions are met. With the eigenvectors, it is possible that no key is found even though one exists. The attacks implementation is realised in Cryptool 2.**

I. Introduction

The Hill Cipher, invented by Lester S. Hill, is based on linear algebra and matrix multiplication for encryption and decryption. Due to the simple matrix multiplication, the method is susceptible to Known-Plaintext Attacks and allows Ciphertext-Only Attacks. By forming plaintext and ciphertext pairs, the key can be calculated in certain cases. The implementation in CrypTool 2 allows an automated calculation of large key matrices.

II. Hill Cipher

For the Hill Cipher, an alphabet is defined, where each letter is assigned an integer number. The alphabet length defines the modulo value m. The key is an $n \times n$ matrix that is invertible modulo m. The plaintext is divided into column vectors based on the alphabet. The rows of the columnvectors have to be equal to the key matrix columns.
The cipher is the result of the multiplication of the key matrix and the plaintext vectors. For the decryption the inverted key is multiplied by the ciphertext vectors [1].

III. Known-Plaintext Attack

A. Introduction

The plaintext and the corresponding ciphertext are known for the Known-Ciphertext Attack. By analyzing these pairs, it is possible to calculate the encryption key used in the cipher. In the context of the Hill cipher, this involves using linear algebra techniques to solve the key matrix. Once the key matrix is determined, it can be used to decrypt other ciphertexts encrypted with the same key.

B. Implementation

To implement the Known-Plaintext Attack, a system of equations for

$$P \cdot K \equiv C \mod m$$

must be created to calculate K. Instead of an equation system, the equation can also be rearranged to

$$P^{-1} \cdot C \equiv K \mod m$$

This way, only the plaintext matrix inverse [2] needs to be calculated, and by multiplying the plaintext matrix with the ciphertext matrix, the key can be obtained. For the implementation, a matrix class as shown in Figure 1 was created. This class contains matrix-specific functions, such as matrix multiplication, determinant and inverse calculation.

```
public class HillCipherAttackMatrix
{
    public int Rows { get; set; }
    public int Cols { get; set; }
    public int[,] Data { get; set; }
    ...
}
```

Figure 1: HillCipherAttackMatrix class

For the first prototype, a console application was created that implements the Gaussian algorithm. After embedding the prototype as a plugin in Cryptool 2, it became apparent that this approach leads to high runtimes and it is complex to maintain. An alternative is offered by using the adjugate matrix or eigenvalues and eigenvectors.
The Cryptool 2 application enables the possibility to implement settings for the individual adjustment of parameters in a user interface. General settings that are applicable to both attacks and attack-specific settings can be adjusted. The settings fundamentally determine which attack is applied and which alphabet is used. In addition, methods are implemented in *HillCipherAttackUtils*, which are used in the Known-Plaintext Attack but are also significant for the Ciphertext-Only Attack. A part of this class includes, for example, the assignment of letters to numbers, the conversion between matrices and vectors and vice versa or some methods for verification.
Generally, it is also determined at which dimension the

calculation should start. This narrows down the search space if something is known about the dimension; otherwise, the calculation starts by default with the 2nd dimension. Futhermore for both attacks, the calculation method can also be determined. The choice consists of specifying the method of the inverse method, which includes either the method with the adjugate matrix or the method with eigenvalues and eigenvectors.

1) Adjugate Matrix: The adjugate matrix also known as the classical adjoint is the transposed cofactor matrix of a $n \times n$ matrix [4]. To rephrase, the adjugate matrix is formed by exchanging the cofactor matrix rows and columns. The Cofactor of an Matrix element is calculated by removing the row and the column to retrieve the submatrix. Then calculate determinant of the submatrix and multiply it by $(-1)^{i+j}$ where i and j are the matrix row and the column. In Figure 2 is the adjugate matrix calculation method displayed and according to the adjugate calculation in Figure 3 is the cofactorcalculation in detail represented.

```
public void CalculateAdjugate(int[,] adj)
{
    int n = Rows;
    if (n == 1)
    {
        adj[0, 0] = 1;
        return;
    }
    int sign;
    HillCipherAttackMatrix temp = new
        HillCipherAttackMatrix(n, n);
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            GetCofactor(temp.Data, i, j, n);
            sign = ((i + j) \% 2 == 0) ? 1 : -1;
            adj[j, i] = (sign * temp.Determinant(
                n - 1));
        }
    }
}
```

Figure 2: CalculateAdjugate method

The determinant calculation as shown in the CalculateAdjugate method, is implemented as a separate method, as it is used in various functions. Among other things, it is used to determine whether the calculation of the inverse is possible or whether a key is valid.

```
public void GetCofactor(int[,] temp, int p,
    int q, int n)
{
    int i = 0, j = 0;
    for (int row = 0; row < n; row++)
    {
        for (int col = 0; col < n; col++)
        {
            if (row != p && col != q)
            {
                temp[i, j++] = Data[row, col];
                if (j == n - 1)
                {
                    j = 0;
                    i++;
                }
            }
        }
    }
}
```

Figure 3: GetCofactor method

2) Eigenvalues and Eigenvectors: The basis for this approach is that the eigenvalues of a matrix A are also the eigenvalues of the inverse of A. The *MathNet.Numerics.LinearAlgebra* library is used for the eigenvalue/eigenvector calculation. This calculation is designed to compute the eigenvalues and eigenvectors in real or complex numbers [3]. As a result, rounding errors may occur and a key cannot be calculated.
Figure 4 is showing how the library is used to calculate the eigenvectors. According to the eigenvectors calculation Figure 5 shows the eigenvalues calculation.

```
public HillCipherAttackMatrix
    InverseByEigenVectors(int mod)
{
    var matrix = Matrix<double>.Build.
        DenseOfArray(ConvertToDoubleArray());
    var (eigenvalues, eigenvectors) =
        CalculateEigenValues(matrix, mod);

    // Calculation of the inverse of the
        eigenvalues
    var invEigenvalues = eigenvalues.Map(x =>
        ModInverseDouble(x, mod));

    // DiagonalMatrix of the inverse
        eigenvalues
    var invEigenvaluesMatrix = Matrix<double>.
        Build.DenseDiagonal(eigenvalues.Count,
        eigenvalues.Count, (i) =>
        invEigenvalues[i]);

    // Calculation of the inverse matrix
    var invMatrix = eigenvectors *
        invEigenvaluesMatrix * eigenvectors.
        Inverse();

    // Reduction of the inverse matrix modulo m
    invMatrix = invMatrix.Map(x => x % mod);
    invMatrix = invMatrix.Map(x => x < 0 ? x +
        mod : x);

    var result = new HillCipherAttackMatrix(
        Rows, Cols);
    for (int i = 0; i < Rows; i++)
    {
        for (int j = 0; j < Cols; j++)
        {
            result.Data[i, j] = (int)Math.Round(
                invMatrix[i, j]) % mod;
            if (result.Data[i, j] < 0)
            {
```

```
                result.Data[i, j] += mod;
            }
        }
    }

    return result;
}
```

Figure 4: InverseByEigenVectors method

For the eigenvectors calculation, the matrix or the vectors must first be converted into double values. In the present class for matrices (Figure 1), the data is stored as integer values and cannot be processed by the library in this form. After the calculation, the values are mapped back to integer values.

```
private static (Vector<double> eigenvalues,
    Matrix<double> eigenvectors)
    CalculateEigenValues(Matrix<double> matrix
    , int modulus)
{
    // Calculation of the eigenvalues and
        eigenvectors
    var evd = matrix.Evd();

    // Reduction of the eigenvalues in modulo m
    var eigenvalues = evd.EigenValues.Map(x =>
        x.Real % modulus);
    eigenvalues = eigenvalues.Map(x => x < 0 ?
        x + modulus : x);

    // Reduction of the eigenvectors in modulo
        m
    var eigenvectors = evd.EigenVectors.Map(x
        => x % modulus);
    eigenvectors = eigenvectors.Map(x => x < 0
        ? x + modulus : x);

    return (eigenvalues, eigenvectors);
}
```

Figure 5: CalculateEigenValues method

Before retrieving the inverse it must be determined what dimension is involved, as an inverse can only be calculated if the matrix is square and the determinant is not zero and coprime to m. Since the dimension is usually unknown, an attempt is made to find a key for each dimension as long as the ciphertext provides enough data for a dimension. If the plaintext is shorter than the ciphertext, "A" is added until the plaintext length is the same as the ciphertext length.
For a dimension n, square matrices are created for plaintext and ciphertext from their vectors. It may happen that for some vectors of the plaintext there is no invertible matrix for a dimension. In such cases, another vector from plaintext and ciphertext is always added to include all possibilities of a dimension, in case no suitable key was found previously.

Once a potential key is found, it is checked whether it is invertible and whether encrypting the plaintext results in the complete ciphertext.

For the testing purpose of the calculation, plaintext and ciphertext pairs were created from dimension 1 to dimension 10. It was important to use plaintexts that are invertible with modulo m in the corresponding dimension. It turned out that the identity matrix of the respective dimension is suitable as a test object. Corresponding keys also had to be created, which, by definition, must also be invertible. With the version of the Hill Cipher implemented in Cryptool 2, the examples could be adjusted and verified.
For the application of the Known-Plaintext Attack, there is a template with the correct input and output windows, as well as a brief guide on how to use the component for this attack. The input windows are filled with example values that lead to a result.

C. Evaluation

To compare the two calculation methods, the runtime for the calculation is measured. Each measurement starts with the first dimension. The time is measured in milliseconds. The average of ten runs per dimension is compared. The values of the time measurement are shown in the Table I.

TABLE I: Comparison of the runtimes regarding the calculation methods

$n \times n$	**Adjugate Matrix**	**Eigenvectors and -values**
1×1	0 ms	0 ms
2×2	0 ms	4 ms
3×3	5 ms	6 ms
4×4	10 ms	14 ms
5×5	15 ms	16 ms
6×6	20 ms	17 ms
7×7	35 ms	20 ms
8×8	141 ms	49 ms
9×9	940 ms	205 ms
10×10	9531 ms	1488 ms

From the comparison, it can be seen that the calculation in higher dimensions is significantly faster with eigenvectors. However, it should be noted that with eigenvectors, there is no guarantee to find a solution, as calculations are performed with double values.
In contrast, the calculation with the adjugate matrix finds a solution if the plaintext matrix and the key are invertible.

IV. Ciphertext-Only Attack

A. Introduction

In a Ciphertext-Only Attack, only the ciphertext is known. To calculate the key, a plaintext must be generated. There are various methods for generating the plaintext. In this case a dictionary is used for generation, which results in a dictionary attack. This offers the advantage of using individual or topic-based dictionaries. It should be noted that the content of the dictionaries must be passed as an array and words less then 5 letters are sorted out. The reason for the minimum length of 5 is to exclude articles and pronouns.

B. Implementation

For the execution of the Ciphertext-Only Attack, this is selected in the settings, and the language to be used is set. The language setting and the threshold setting are used to evaluate the calculated keys. Depending on the language, the frequencies of letters, bigrams and trigrams differ. The threshold indicates how much the ciphertext from the calculated valid key and the plaintext may differ from the given ciphertext. The measurement is performed using the Levenshtein distance [5].
For each key that has been calculated and meets the requirements of generating a ciphertext within the Levenshtein distance as in Figure 6 and being invertible, an entry is created in the presentation interface.

```
private bool CompareCipherTextWithTreshhold(
    HillCipherAttackMatrix key,
    HillCipherAttackMatrix[] plain_mats,
    Dictionary<string, int> alphabet_numbers,
    int treshold)
{
   var _cipherText = Encrypt(key, plain_mats,
       alphabet_numbers);
   if (_cipherText == null)
   {
      return false;
   }

   // uses Levenshtein Distance with treshhold
   int sourceLength = Cipher.Length;
   int targetLength = _cipherText.Length;

   int[,] distance = new int[sourceLength + 1,
        targetLength + 1];

   for (int i = 0; i <= sourceLength; distance
       [i, 0] = i++) { }
   for (int j = 0; j <= targetLength; distance
       [0, j] = j++) { }

   for (int i = 1; i <= sourceLength; i++)
   {
      for (int j = 1; j <= targetLength; j++)
      {
         int cost = (_cipherText[j - 1] ==
             Cipher[i - 1]) ? 0 : 1;
         distance[i, j] = Math.Min(
            Math.Min(distance[i - 1, j] + 1,
                distance[i, j - 1] + 1),
            distance[i - 1, j - 1] + cost);
      }
   }

   return distance[sourceLength, targetLength]
        <= treshold;
}
```

Figure 6: CompareCipherTextWithTreshhold method

The higher the Levenshtein distance is set, the more can potentially be found; however, the solutions will correspondingly be less accurate. With a value of 0, it is expected that the calculated key will reproduce the ciphertext, which may result in no solution being found.

```
internal static double CalculateScore(string
    plaintext, Dictionary<string, int>
    alphabet, int language)
{
   double score = 0;
   // English
   var letterFrequency =
       HillCipherAttackRessources.
       LetterFrequenciesEN;
   var bigramFrequency =
       HillCipherAttackRessources.
       BigramFrequenciesEN;
   var trigramFrequency =
       HillCipherAttackRessources.
       TrigramFrequenciesEN;
   if (language == 1)
   {
      // German
      bigramFrequency =
          HillCipherAttackRessources.
          BigramFrequenciesDE;
      trigramFrequency =
          HillCipherAttackRessources.
          TrigramFrequenciesDE;
      letterFrequency =
          HillCipherAttackRessources.
          LetterFrequenciesDE;
   }
   // Calculates the score for the plaintext
       with the alphabet from the settings and
       the frequencies of the language in the
       Ressources
   foreach (char c in plaintext)
   {
      if (alphabet.ContainsKey(c.ToString()))
      {
         score += letterFrequency[c];
      }
   }
   for (int i = 0; i < plaintext.Length - 1; i
       ++)
   {
      string bigram = plaintext.Substring(i,
          2);
      if (bigramFrequency.ContainsKey(bigram))
      {
         score += bigramFrequency[bigram];
      }
   }
   for (int i = 0; i < plaintext.Length - 2; i
       ++)
   {
      string trigram = plaintext.Substring(i,
          3);
      if (trigramFrequency.ContainsKey(trigram
          ))
      {
         score += trigramFrequency[trigram];
      }
   }
   return (score * 100)/ plaintext.Length;
}
```

Figure 7: CalculateScore method

The presentation interface contains information about the selected alphabet, ciphertext, and the modulo value. A successfully calculated key is entered into the leaderboard, for which a score is calculated as shown in Figure 7. The score calculation includes resources that indicate the frequencies of letters, bigrams, and trigrams. A distinction is made between the German and English languages. Each entry implemented as a dictionary is assigned a percentage value based on its frequency. The letter and bigram frequencies are based on online sources [6] and vary from language to language. Those for German and English are applied and stored in the *HillCipherAttackRessources* class.

The list is sorted in descending order based on this score. The first entry in this list is the best result based on the score, and its key is used as the output. For testing the Ciphertext-Only Attack, a significantly reduced version of a dictionary was used, which was embedded in the code to keep the execution time manageable.

For the Ciphertext-Only Attack application, there is a template with the correct input and output windows, as well as a brief guide on how to use the component for this attack. Here, one input window is a dictionary. The input window for the ciphertext is filled with an example that leads to a result.

V. Conclusion

During the creation of the attacks for Cryptool 2, it turned out that the intuitive implementation with the Gaussian algorithm did not lead to the desired result. The alternative implementation with the adjugate matrix or with the eigenvectors simplified the inverse calculation clearer and increased the efficiency. Using the library for eigenvalues and eigenvectors was a way to achieve a result within the scope of this time-limited project. For the revision or to ensure that a solution can be found just as reliably with this approach as with the adjugate matrix, a calculation of the eigenvalues or eigenvectors with integers in modulo m should be implemented.

A future extension can be the enhancement of the score calculation based on frequencies and the Levenshtein distance. Furthermore regarding the Ciphertext-Only Attack other analyses for the generated plaintext can also be introduced. It may be useful to integrate large language models or to expand the analysis to other languages.

References

[1] Wikipedia contributors, "Hill cipher," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Hill_cipher (last edited October 17, 2024)

[2] Wikipedia contributors, "Invertible Matrix," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Invertible_matrix (last edited December 16, 2024).

[3] Winfried Just, *Eigenvectors and Eigenvalues of Inverse Matrices and Matrix Transposes*, Department of Mathematics, Ohio University, MATH3200: Applied Linear Algebra, 2024.

[4] Wikipedia contributors, "Adjugate Matrix," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Adjugate_matrix (last edited November 15, 2024).

[5] Wikipedia contributors, "Levenshtein distance," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Levenshtein_distance (last edited August 28, 2024)

[6] Wikipedia contributors, "Letter frequency," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Letter_frequency (last edited December 19, 2024)