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

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

***Abstract*—This project implements the Known-Plaintext Attack and the Ciphertext-Only Attack for the Hill Cipher 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 length of the alphabet 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 so. The rows of the columvectors have to be equal to the columns of the key matrix.// 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 th vectors of the ciphertext.

III. Known-Plaintext Attack

A. Introduction

For the Known-Plaintext Attack the plaintext and its corresponding ciphertext is available. 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 for 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 the system of equations, the equation can also be rearranged to

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

This way, only the inverse of the plaintext matrix needs to be calculated, and by multiplying it 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 user interface. General settings that are applicable to both attacks and attack-specific settings can be adjusted. The settings fundamentally determine which attack is to be applied and which alphabet is to be used.
Generally, it is also determined at which dimension the calculation should start. This serves to narrow 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 transpose of the cofactor matrix of a $n \times n$ matrix . To rephrase, the adjugate matrix is formed by exchanging the rows and columns of the cofactor matrix. The Cofactor of an Matrix element is calculated by removing the row and the column to get the submatrix. Then calculate determinant of the submatrix and multiply it by $(-1)^{i+j}$ where i and j are the row and the column of the element. In Figure 2 is the calculation method of the adjugate matrix 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

```
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. 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 calculation of the eigenvalues.

```
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

```
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.
For a dimension n, square matrices are created for plaintext and ciphertext from their vectors. It may happen that some vectors of the plaintext do not create an 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 ciphertext, not just a part of it.

C. Evaluation

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

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.

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 bigrams, trigrams, etc., 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 done using the Levenshtein distance.

```
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

```
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

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.
The presentation interface contains information about the selected alphabet, ciphertext, and the modulo value. A successfully calculated key is entered into the so-called 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 list is sorted in descending order based on this score. The first entry in this list, i.e., the best result, is taken, and its key is used as the output.

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