Relatório - Cifra de Vigenère

Daltro Oliveira Vinuto - 160025966

1. Introdução

Esse trabalho tem como objetivo implementar na linguagem C++ no sistema operacional Linux as operações de cifração, decifração e ataque de recuperação de senha sobre a Cifra de Vigenère. A Cifra de Vigenère[1] foi criada em 1553 e quebrada somente em 1863. Essa cifra consiste na aplicação de várias cifras de César com letras diferentes. As implementações foram feitas usando o material disponibilizado em sala de aula e em conteúdos online.

2. Codificação

A codificação[7] implementada segue os seguintes passos:
1. No momento de execução são fornecidos os arquivos de plain text com o texto sem nenhuma codificação, o arquivo de hidden text que conterá o arquivo após codificação e a linguagem a ser utilizada(inglês ou português).
2. Uma chave é requisitada ao usuário.
3. Abrimos e lemos o arquivo plain text.
4. Alinhamos a chave repetidas vezes até o fim do texto com as letras do texto a ser codificado e somamos a letra a ser codificada com a letra correspondente na chave e aplicamos o módulo 26. A soma resultante determinará a letra a ser codificada.
5. O texto já codificado é salvo no arquivo hidden text.

Segue trecho do código abaixo:

```
//printVector(lines);
// Encrypting ----------------------------------
unsigned k = 0;
for(unsigned i = 0; i < lines.size(); i++) {
    std::string line = lines[i];
    std::string new_line = "";
    for(unsigned j = 0; j < line.size(); j++) {
        char letter = line[j];
        char new_letter = letter;

        if (letter >= 97 && letter <= 122) {
            //printf("d d d\n", letter, new_letter, key[k]);
            //printf("c c c\n", letter, new_letter, key[k]);
            // ASCII code for 'a' is 97
            new_letter = ((letter-'a')+(key[k]-'a'))%26 + 'a';

            k = (k+1)%key.size();

            //printf("d d d\n", letter, new_letter ,key[k]);
            //printf("c c c\n", letter, new_letter, key[k]);
        }


        new_line+= new_letter;

    }
    aux_lines.push_back(new_line);

}
```

3. Decodificação

A decodificação[7] implementada segue os seguintes passos:
1. No momento de execução são fornecidos os arquivos de plain text com o texto sem nenhuma codificação, o arquivo de hidden text que conterá o arquivo após codificação e a linguagem a ser utilizada(inglês ou português).
2. Uma chave é requisitada ao usuário.
3. Abrimos e lemos o arquivo hidden text.
4. Alinhamos a chave repetidas vezes até o fim do texto com as letras do texto codificado e subtraímos a letra a ser decodificada da letra correspondente na chave e aplicamos o módulo 26. A diferença resultante determinará a letra a ser decodificada.
5. O texto já decodificado é salvo no arquivo plain text.

Segue trecho do código abaixo:

```
// Decrypting ----------------------------------
unsigned k = 0;
for(unsigned i = 0; i < lines.size(); i++) {
    std::string line = lines[i];
    std::string new_line = "";
    for(unsigned j = 0; j < line.size(); j++) {
        char letter = line[j];
        char new_letter = letter;

        if (letter >= 97 && letter <= 122) {
            //printf("d d d\n", letter, new_letter, key[k]);
            //printf("c c c\n", letter, new_letter, key[k]);
            // ASCII code for 'a' is 97
            new_letter = ((letter-'a')-(key[k]-'a'))%26;
            if (new_letter < 0) {
                new_letter+= 26;
            }
            new_letter+=  'a';

            k = (k+1)%key.size();

            //printf("d d d\n", letter, new_letter ,key[k]);
            //printf("c c c\n", letter, new_letter, key[k]);
        }
```

4. Ataque

O ataque[2,3,4] implementado segue os seguintes passos:
1. No momento de execução são fornecidos os arquivos de plain text com o texto sem nenhuma codificação, o arquivo de hidden text que conterá o arquivo após codificação e a linguagem a ser utilizada(inglês ou português).
2. Descobrimos o tamanho da chave.
3. Descobrir as letras que compõem a chave de um certo tamanho.
4. Chamamos a função já implementada para decodificar uma vez que nesse momento já encontramos a chave.
5. Perguntamos ao usuário se a decodificação foi bem sucedida se sim então terminamos o ataque caso o contrário mostramos para o usuário a lista da frequência dos fatores de trigramas e pedimos para o usuário escolher um tamanho de chave e daí voltamos para passo 3.

4.1 Descobrindo o tamanho da chave

Para descobrir o tamanho da chave foi feita a contagem de trigramas presentes no texto codificado. Para todos os trigramas presentes no texto codificado são contadas a distância entre dois trigramas iguais e os fatores dessa distância são salvos. O programa tentará usar como tamanho da chave o fator com mais repetições, porém se isso falhar então esses fatores serão mostrados para que o usuário escolha um tamanho de chave dentre os que possuem mais repetições até que o texto decodificado seja legível.

Segue trecho do código abaixo:

```
unsigned Vigenere::FindKeyLength(const std::vector<char>& letters) {
    unsigned length = 1;

    //std::cout << "letters: "; printVector<char>(letters);

    unsigned maximum_length = 30;
    std::vector<unsigned> factors;
    for(unsigned i = 0; i < maximum_length; i++) {
        factors.push_back(0);
    }

    for(unsigned i = 0; i < letters.size()-2 ; i++) {
        std::string first_trigram = "";
        first_trigram += letters[i];
        first_trigram += letters[i+1];
        first_trigram += letters[i+2];
        //std::cout << first_trigram << " , ";
        for(unsigned j = i+3; j < letters.size()-2; j++) {
            std::string next_trigram = "";
            next_trigram += "";
            next_trigram += letters[j];
            next_trigram += letters[j+1];
            next_trigram += letters[j+2];

            if (next_trigram == first_trigram) {
                unsigned distance;
                distance = j - i;

                addFactors(factors, distance);
            }
        }
    }


    unsigned max_value = 0;
    for(unsigned i = 0; i < factors.size(); i++) {
        //std::cout << "factor[i]: " << factors[i] << std::endl;
        if (factors[i] > max_value) {
            max_value = factors[i];
            length = i+2;
            //std::cout << max_value << " , " << length << std::endl;
        }
    }

    //std::cout << "key length: " << length << std::endl;

    //std::cout << "factors: "; printVector<unsigned>(factors," , ");
    this->factors = factors;

    return length;
}
```

4.2 Análise de Frequência.

Para realizar a análise da frequência, considerando n como o tamanho da chave então inicialmente dividimos o texto codificado em n subtextos, cada subtexto corresponde a uma cifra de césar, isso é, ele está deslocado em apenas uma letra então realizamos o ataque de análise de frequência em cada subtexto. Para isso utilizamos a frequência do idioma escolhido e a frequência das letras dentro do subtexto. Deslocamos esse idioma para direita 26 vezes e armazenamos os vetores resultantes, de posse desses vetores fazemos o produto escalar de cada vetor deslocado como vetor da frequência da letras dentro do subtexto e escolhemos o que o vetor que resultou no maior produto escalar, então utilizamos o deslocamento que foi aplicado para gerar o vetor como a chave procurada.

Segue trecho de código:

```
// fill the matrix
for(unsigned i = 0; i < key_length; i++) {

    for(unsigned j = i; j < letters.size(); j+=key_length) {
        char letter = letters[j];
        matrix[i].push_back(letter);
    }
}

//printMatrix<char>(matrix);
//printVector<char>(matrix[0]);

// Find the key of each vector(line of the matrix)
// using the BreakCaesarCipher method
key = "";
for(unsigned i = 0; i < key_length; i++) {
    key+= BreakCaesarCipher(matrix[i]);
}
```

```
for(unsigned i = 0; i < alphabet_size; i++) {
    for(unsigned j = 0; j < alphabet_size; j++) {
        double value;
        unsigned k;
        k = (26-i + j)%26;
        value = statistic_vector[k];
        matrix[i][j] = value;
    }
}
```

```
double max_value = 0;
unsigned index = 0;
for(unsigned i = 0; i < matrix.size(); i++) {
    double product;

    product = GetProduct(matrix[i], values_vector);

    if (product > max_value) {
        max_value = product;
        index = i;
    }
}

key = (index) +'a';
```

5. Conclusão

Com os computadores atuais fica mais fácil quebrar a cifra de Vigenère[1] se as chaves forem pequenas porém se a chave for grande suficiente e sua

implementação adequada então a técnica se torna mais segura uma vez que teremos subtextos menores e assim a chance do ataque por análise de frequência funcionar diminui.
Foi muito interessante realizar esse experimento de criptoanálise pois verificamos na prática um dos processos usados para quebra de cifras e aprendemos que analise matemática em especial a estatística é essencial para a criptoanálise.
A principal dificuldade durante a implementação foi descobrir o tamanho da chave uma vez que foram usadas algumas técnicas estatísticas e matemáticas e pela própria grande variedade de técnicas diferentes para encontrar esse tamanho. Encontramos o tamanho da chave contando trigramas porém talvez usando outra técnica como Índice de coincidência(5) ou Fitness measure(6) permitisse encontrar o tamanho com mais precisão.

6. Referências

1. Wikipedia (2023) . Vigenère Cipher.
https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher

2. Youtube. 2014. Cryptography - Breaking the Vigenere Cipher.
https://www.youtube.com/watch?v=P4z3jAOzT9I&t=106s&ab_channel=BrianVeitch

3. Youtube. 2020. Cryptanalysis of Vigenere cipher: not just how, but why it works.
https://www.youtube.com/watch?v=QgHnr8-h0xI&t=773s&ab_channel=ProofofConcept

4. Youtube. 2015. Vigenere Cipher - Decryption(Unknown Key).
https://www.youtube.com/watch?v=LaWp_Kq0cKs&t=289s&ab_channel=Theoretically

5. Five Ways to Crack a Vigenere Cipher.
https://www.cipherchallenge.org/wp-content/uploads/2020/12/Five-ways-to-crack-a-Vigenere-cipher.pdf

6. Quadgram Statistics as a Fitness Measure.
http://practicalcryptography.com/cryptanalysis/text-characterisation/quadgrams/

7. Vigenere Cipher.
https://crypto.interactive-maths.com/vigenegravere-cipher.html