Segurança Computacional

Trabalho 1

Gabriel Nogueira - 18/0113330

1 Cifrador/Decifrador

A cifra de Vigenère é uma variação da cifra de César, que assume que todos os caracteres do alfabeto A utilizado são codificados por um número. Para criptografar uma mensagem M com uma chave K, a mensagem criptografada C é obtida pela equação:

$$C_i = (M_i + K_i) \mod len(A)$$

onde C_i, M_i e K_i representam o código do i-ésimo caractere contido no texto e $len(A)$ representa o tamanho do alfabeto escolhido. Da mesma forma, a mensagem original pode ser obtida por meio da subtração da chave aplicada à mensagem criptografada:

$$M_i = (C_i - K_i) \mod len(A)$$

Para a implementação de um cifrador/decifrador, foram criadas as funções `encrypt_char` e `decrypt_char`, que implementam a cifra de César para um caractere, somando ou subtraindo o valor da chave ao valor do caractere fornecido e retornando esse valor no espaço modular do tamanho do alfabeto (representado pela lista `CHARACTERS`). Para isso, foram utilizadas as funções `code` e `decode`, respectivamente responsáveis por retornar o código de um caractere e por retornar o caractere correspondente a um código.

É importante ressaltar que as funções `encrypt_char` e `decrypt_char` ignoram caracteres que não estão contidos no alfabeto escolhido, de forma a não criptografá-los ou decifrá-los. Isso é feito para evitar erros e manter a integridade da mensagem original.

```
def encrypt_char(m, k):
    return (decode((encode(m) + encode(k))%len(CHARACTERS))
      if m in CHARACTER_CODES else m)
```

```
def decrypt_char(m, k):
    return (decode((encode(m) - encode(k))%len(CHARACTERS))
    if m in CHARACTER_CODES else m)
```

Após a implementação das funções `encrypt_char` e `decrypt_char`, foi possível elaborar as funções encrypt e decrypt para implementar a cifração e decifração de uma mensagem com a cifra de Vigenère. Para isso, foi utilizada a função `cycle` da biblioteca padrão `itertools` da linguagem Python. Essa função é capaz de criar um iterador circular que retorna sempre o próximo caractere da chave K, o que evita a necessidade de expandi-la ou comprimi-la até que tenha o mesmo tamanho da mensagem M.

```
def encrypt(msg, key):
    key_it = cycle(key)
    return ''.join([encrypt_char(m,next(key_it))
    if m in CHARACTER_CODES else m for m in msg])

def decrypt(msg, key):
    key_it = cycle(key)
    return ''.join([decrypt_char(m,next(key_it))
    if m in CHARACTER_CODES else m for m in msg])
```

Dessa forma, para cada caractere contido na mensagem, as funções encrypt e decrypt aplicam o processo de cifração ou decifração com o devido caractere da chave, ou simplesmente ignoram o caractere caso ele não esteja no alfabeto utilizado. Isso garante que a mensagem original seja mantida intacta, sem a introdução de caracteres desnecessários ou indesejados.

2 Ataque de recuperação de senha

Embora mais complexa do que a cifra de César, a cifra de Vigenère ainda é vulnerável a métodos de criptoanálise, como o método de análise de frequências. No entanto, para quebrar a cifra é necessário saber em que idioma a mensagem original está escrita.

O primeiro passo do método é encontrar o tamanho l da chave utilizada na cifração, para isso foi realizada uma análise de trigramas, em que todos os trigramas da mensagem cifrada são examinados. Quando um trigrama se repete, é calculada a distância s entre as duas ocorrências desse trigrama. A

repetição de um trigrama na mensagem cifrada pode indicar a repetição de um trigrama na mensagem original e, além disso, que os mesmos caracteres utilizados para cifrar a ocorrência anterior estão sendo utilizados para cifrar a atual. Nesse sentido, todos os divisores inteiros positivos de s são possíveis candidatos para o tamanho da chave.

```
Key:        ABCDABCDABCDABCDABCDABCDABCD
Plaintext:  cryptoisshortforcryptography
Ciphertext: CSASTPKVSIQUTGQUCSASTPIUAQJB
```

No exemplo apresentado acima, o trigrama "CSA" aparece na posição 0 e, em seguida, na posição 16 da mensagem cifrada. Caso a chave utilizada seja repetida no texto da mensagem, isso indicaria que a chave deveria recomeçar a partir da posição 16. Como resultado, os possíveis tamanhos da chave seriam 1, 2, 4, 8 ou 16, que são os divisores inteiros positivos de 16, a diferença entre as duas ocorrências do trigrama. Assim, para cada trigrama encontrado na mensagem cifrada, esse procedimento é repetido, e os divisores mais frequentes são marcados como os possíveis tamanhos da chave.

Abaixo, temos a implementação da função que realiza o processo de análise de trigramas descrito anteriormente. Essa função utiliza um objeto do tipo `Counter` como tabela de frequências para os divisores marcados, além do dicionário `pos` para guardar a ultima ocorrência do trigrama encontrado . Ao final do processo, o número 1 é retirado do contador, uma vez que ele será sempre o mais frequente e dificilmente uma chave terá tamanho igual a 1. A função também recebe um parâmetro chamado `n_options`, que determina quantos dos tamanhos de chave mais prováveis serão retornados.

```
def find_key_length(msg, n_options=10 ):
    pos = {}
    lens = Counter()
    for i in range(len(msg)):
      s = msg[i:i+3]
      if s in pos: lens.update(get_divs(i-pos[s]))
      pos[msg[i:i+3]]= i
    del lens[1]
    key_lens = lens.most_common(n_options)
    return key_lens
```

Após descoberto o tamanho l da chave K, é necessário descobrir o valor de cada um dos seus caracteres K_i. Sendo assim, os caracteres da mensagem

M são divididos em l grupos, onde dois caracteres M_i e M_j pertencem ao mesmo grupo se a distância entre os dois é l. Dessa forma, os caracteres presentes do grupo i foram cifrados pelo caractere K_i da chave. Tais grupos da mensagem cifrada são gerados pela função abaixo:

```
def get_groups(msg, key_len):
    return [msg[i::key_len] for i in range(key_len)]
```

Para que seja descoberto o valor de K_i, foi desenvolvida a função `break_char`, ela toma como argumentos os caracteres presentes no grupo e a linguagem em que se acredita estar a mensagem. Em seguida, ela retorna uma lista de possíveis caracteres para K_i em ordem decrescente de probabilidade.

```
def break_char(group, language):
    real_freqs = list(FREQS_PT.values() if language=='PT'
      else FREQS_EN.values())
    txt_freqs = get_freqs(group)
    rank = []
    for c in CHARACTERS:
      rank.append(((sum(abs(a-b) for a, b in
zip(real_freqs, txt_freqs))), c))
      txt_freqs.append(txt_freqs.pop(0))
    return [c for _,c in sorted(rank)]
```

Para que seja obtida a lista de possíveis caracteres para K_i, é feita uma comparação da frequência dos caracteres presentes no grupo i com a frequência dos caracteres presentes na linguagem escolhida. O código cria uma lista vazia chamada rank, que será preenchida com tuplas contendo a diferença entre as frequências reais e as frequências do texto para cada letra do alfabeto, juntamente com a respectiva letra. A diferença é calculada somando-se as diferenças absolutas entre os valores de frequência correspondentes.

Através de um loop que percorre todas as letras em `CHARACTERS` (que representa o alfabeto utilizado), cada tupla é adicionada à lista rank. Antes de adicionar a tupla, a lista `txt_freqs` é rotacionada, movendo o primeiro elemento para o final, o que corresponde a considerar o próximo caractere da chave.

Em seguida, a função retorna uma lista contendo apenas as letras ordenadas de acordo com a diferença calculada, ou seja, as letras que têm maior probabilidade de corresponder ao caractere da chave.

```
def find_key(msg,key_len, language):
```

```
return [break_char(group, language)
for group in get_groups(msg, key_len)]
```

Por fim, a função `find_key` percorre cada grupo de caracteres obtido pela função `get_groups` e chama a função `break_char` para cada grupo. Portanto, o resultado da função será uma lista contendo as letras mais prováveis para cada caractere da chave em cada grupo de caracteres da mensagem criptografada.