Criptografia 🌐 📱

Aluna: Giovanna Fioravante Dalledone

> GRR: 20232370

Cifra Gi-Playfair-Fence

Sumário:

Linguagem: C

1. Processo de Implementação

A implementação do trabalho teve início pela cifra playfair. Antes de tudo, era um código para testes e, quando foi ficando grande demais, foi separado em arquivos e modularizado.

1.1 Playfair

Arquivos:

```
├── playfair.h  --> Protóptipos das funções de manipulação do texto e
cifra/decifra da playfair.
├──── playfair.c  --> Implementação de playfair.h.
├────── cifra.c    --> Utiliza o .c e o .h para cifrar um texto com as
regras da playfair (Rail Fence também).
├──────── decifra.c  --> Utiliza o .c e o .h para decifrar um texto com as
regras da playfair (Rail Fence também).
```

Estruturas:

```c
struct playfair_t {
    char matriz[5][5];              //Matriz que será preenchida
    char *chave;                    //Chave que preenche a matriz (não
repete letras)
    char *chave_recebida;           //Registro da chave passada pelo
usuário
    char *texto_cifrado;            //Texto cifrado e formatado
    char *texto_decifrado;          //Texto decifrado e formatado
    unsigned int tamanho_chave;     //Tamanho da chave para facilitar as
contas
    char linha;                     //Linha alvo para
codificação/decodifucação
```

```
    char coluna;                    //Coluna alvo para
codificação/decodificação
};

struct texto_t{
    unsigned int tamanho;           //Número de caracteres do texto com
espaços
    unsigned int num_pares;         //Quantidade de parzinhos formados ->
não utilizei
    char *texto_base;               //Texto base sem espaços e com letras
de complemento (x)
    char *pares_originais;          //Pares de indexação para codificar o
texto
};

struct alfabeto_t {
    char maiusculas[25];            //contém todas as letras do alfabeto
para facilitar a vida (0 ~ 25)
};
```

O arquivo principal da playfair é `playfair.c` no qual a implementação de cada função é feita. O Projeto foi sofrendo diversas mudanças ao longo da implementação, por exemplo:

- Antes, o alfabeto também guardava as letras minúsculas. Depois eu decidi que seria mais fácil trabalhar somente com um tipo de letra, para melhorar a insersão delas na matriz.
- Depois de algumas experiências não muito boas, optei por utilizar uma espécie de "bitmap" para mapear as letras na matriz. Seu funcionamento é simples: cria-se um vetor auxiliar inicializado com zeros. Conforme lemos a chave, marcamos com 1 a posição correspondente ao índice da letra. O índice é encontrado subtraindo o valor do caractere do valor de `A` (65). Então, as letras da chave e as demais são colocadas ao mesmo tempo na matriz.
- Pensei em usar um `switch case` para implementar as regras da matriz, mas não consegui desenvolver muito bem. Então, mantive a estrutura de `else if`.
- Vale ressaltar que esse código da playfair não insere um X entre duas letras iguais.

O que é passado para o cifrador é o caminho do texto a ser cifrado, exemplo: `diretorio/texto.txt`.
O decifrador, por sua vez, devolve um arquivo `arquivo_decifrado_playfair.txt`.

1.2 Rail Fence

Arquivos:

```
├── rail_fence.h  --> Protóptipos das funções de manipulação do texto e
cifra/decifra da rail_fence.
├──── rail_fence.c  --> Implementação de rail_fence.h.
├────── cifra.c    --> Utiliza o .c e o .h para cifrar um texto com as
regras da Rail Fence (Playfair também).
├─────── decifra.c  --> Utiliza o .c e o .h para decifrar um texto com as
regras da Rail Fence (Playfair também).
```

Estrutura

```
struct rail_fence_t {
    unsigned char **matriz;                 //Matriz que vai conter todos
os caracteres
    unsigned char *texto_limpo;             //Texto base livre de espaços
e acentuações
    unsigned char *texto_cifrado;           //Texto cifrado
    unsigned char *texto_decifrado;         //Texto decifrado
    unsigned long int num_linhas;           //Chave
    unsigned long int num_colunas;          //Inicialmente fixo, mas pode
mudar se houver muitos caracteres
    unsigned long int num_caracteres;       //Número de caracteres do
TEXTO LIMPO
};
```

- Seguindo a mesma lógica da Playfair, a implementação da Rail Fence é parecida. A estrutura foi definida no .h e as devidas implementações foram feitas no .c, além disso, as funções são utilizadas em `cifra.c` e `decifra,c`.
- Depois de uma breve pesquisa sobre desempenho, a forma de tratar o texto escolhida foi alocar uma matriz imensa, em detrimento de tratar o texto por partes. Assim todos os caracteres são colocados de uma vez na matriz e fica muito mais simples o trabalho.
- Essa matriz é montada tanto para cifrar quanto para decifrar, porém, para decifrar precisa-se algumas informações adjacentes como a quantidade de linhas de colunas que o cifrador utilizou, por isso estão separadas em duas funções diferentes.
- Eu tentei não montar a matriz para decifrar, mas isso se mostrou uma tarefa incrivelmente mais complicada, tanto que o resultado saiu bem errado.
- Para cifrar, o algoritmo fixa uma coluna e percorre todas as linhas, escrevendo-as em um arquivo. A decifragem funciona da mesma maneira, após remontar a matriz, basta fixar uma coluna e percorrer todas as linhas até o fim de todas as colunas.
- Depois de implementar a Playfair, foi consideravelmente mais fácil pensar e implementar a Rail Fence. Talvez ela seja mais simples de implementar mesmo, mas a impressão deixada foi que a parte mais difícil do trabalho foi limpar e tratar o texto. Com isso pronto, o restante fluiu rápido e bem.
- Um ponto talvez não tão trivial foi corrigir os bugs que surgiram pelo caminho

1.3 AES

Implementar o AES foi mais difícil do que parecia. Para essa cifra, foi criado mais um par de arquivos `.c`e `.h` para cifrar e decifrar o texto, junto com uma função auxiliar de gerar bytes aleatporios.

Principais Erros

1. Disparado, o principal problema foi entender o fluxo das funções da documentação. Depois disso, ficou mais simples entender o que precisava ser feito: `criar o contexto de criptografia` -> `inicializar o contexto criado` -> `cifrar em blocos de bytes` -> `tratar o último bloco` -> `liberar o contexto de criptografia`. Descobri da pior maneira que não existe uma função em `C` que encapsule isso tudo.

2. O segundo pior problema foi tratar a chave. O formato dela causou certa confusão no momento de imprimir no terminal e também ler do terminal. A solução foi tratar a chave lida (em `decifra.c`) de hexadecimal para binário.
3. Pelo incrível que pareça, nos primeiros testes da cifragem eu utilizei a chave errada e isso gerou grandes problemas até que eu pudesse entender o que estava acontecendo.

2. Arquivos

O código gera muitos arquivos, abaixo segue a ordem deles:

`Cifra`

1. `Texto claro`: Entra na Playfair.
2. `arquivo_cifrado_playfair.txt`: Entra na Rail Fence.
3. `arquivo_cifrado.txt`: Texto final cifrado.

`Decifra`

1. `arquivo_cifrado.txt`: Entra na Rail Fence.
2. `arquivo_decifrado_rf.txt`: Parcialmente decifrado, entra na Playfair.
3. `arquivo_decifrado.txt`: Arquivo final decifrado.

> OBS: O arquivo final decifrado vai ser diferente do texto original por doi motivos: o código não codifica os caracteres especiais multibyte como os símbolos, apenas os ignora, e também não trata os casos das letras acentuadas ou caracteres como `ç`, eles são apenas "engolidos".

`Arquivos adicionais`

1. `Gráficos`: Gerados pelo script em pyhton.
2. `Textos`: Em um diretório separado.

3. Resultados

O algoritmo `AES` é absurdo e performa incrivelmente bem em todos os testes. O algoritmo que implementei demora para tratar o texto e tem uma cifragem relativamente rápida pois trabalha quase sempre com matrizes e indexações simples.
Dois fatores limitadores para o desempenho do algoritmo podem ser:

1. A capacidade do computador, pois, quando testei no computador do laboratório LIAMF o algoritmo rodou quase que instantaneamente para o texto entre `100KB` e `1MB`, além de levar um minuto ou dois para cifrar o arquivo maior do que `1MB`. Entretanto, no laptop pessoal no qual testei, o ambos os arquivos levaram mais do que 5 minutos para rodar.
2. O fato de não processar em blocos. O código tem uma característica de tratar o texto em um vetor absurdo e enorme na Playfair e uma matriz imensa na Rail Fence. Ao invés de tratar o texto em blocos menores, como faz o AES, optei por algo mais direto, pois, em um bom computador não é para afetar tanto no desempenho e a implementação ficou mais simples.
3. Um dos motivos que fazia o algoritmo demorar para ser executado era a forma com que eu estava gravando os dados no arquivo da playfair, demorava demais porque eu precisava abrir o arquivo e chamar o sistema operacional muitas vezes. A solução foi, simplesmente, aproveitar o super buffer que eu coloquei o texto e escrever tudo de uma vez só. É nítido o quanto isso melhorou o

desempenho, antes dessa melhoria, o código demorava mais do que cinco minutos para textos médios e grandes.