Ciências ULisboa | Informática

Arquiteturas de Sistemas Computacionais

Projeto

Esteganografia e Criptografia em *Assembly*

---

1 Introdução

1.1 O que são a esteganografia e a criptografia?

A esteganografia é uma área que estuda diversas técnicas para ocultar a **existência** de uma mensagem dentro de outra. Ela difere da criptografia, pois esta última foca-se no estudo de técnicas para ocultar o **significado** de uma mensagem. As duas áreas complementam-se, pois, um indivíduo pode querer transmitir uma mensagem escondida em outra (i.e., esteganografia) ao mesmo tempo que esta mensagem será enviada cifrada (i.e., criptografia). Neste caso, mesmo que alguém consiga intercetar a mensagem oculta, não conseguirá ler o seu conteúdo sem decifrá-la.

1.2 Exemplos práticos de esteganografia e criptografia

Em meios digitais, o conteúdo de uma mensagem pode ser de qualquer tipo de dados que possa ser representado através de dígitos binários. Textos, documentos, imagens, áudios e vídeos são exemplos comuns de tipos de dados enviados em mensagens.

Relativamente ao uso conjunto de esteganografia e criptografia em meios digitais, um exemplo comum consiste em **esconder uma mensagem de texto cifrada numa imagem**, o qual será o tema deste projeto. Outros exemplos, ortogonais a este projeto, incluem esconder textos e imagens em músicas por diversão; marcas d'água em imagens e vídeos para a proteção de direitos de autor e de distribuição; textos subliminares em notícias para contornar meios de comunicação que estejam a ser vigiados por inimigos ou sob censura, etc.

2 Contexto

2.1 Exemplo de um fluxo de execução

Para contextualizar os objetivos de esconder uma mensagem de texto cifrada numa imagem e recuperar esta mensagem, é apresentado o fluxo de execução da Figura 1. Neste exemplo, um *Remetente* escreve uma *Mensagem Original* num ficheiro de texto e escolhe uma *Rotação r* para cifrar a mensagem e uma *Imagem Original* onde a *Mensagem Cifrada* será escondida. O *Remetente* executa então um programa, chamado *Esconder*, o qual:

E1. Verifica se recebeu a quantidade correta de argumentos e imprime uma mensagem de erro em caso negativo.

E2. Aplica o algoritmo de criptografia (uma rotação para a direita de *r* vezes, `ror r`) para cifrar cada caractere da *Mensagem Original* de texto, gerando a *Mensagem Cifrada*;

E3. Aplica a esteganografia para esconder a *Mensagem Cifrada* na *Imagem Original*, gerando a *Imagem Modificada*.

O programa *Esconder* receberá apenas quatro argumentos, nesta ordem: o caminho para o ficheiro da *Mensagem Original* de texto, a *Rotação r*, o caminho para o ficheiro da *Imagem Original* em BMP e o caminho onde deverá ser escrito o ficheiro da *Imagem Modificada* em BMP.

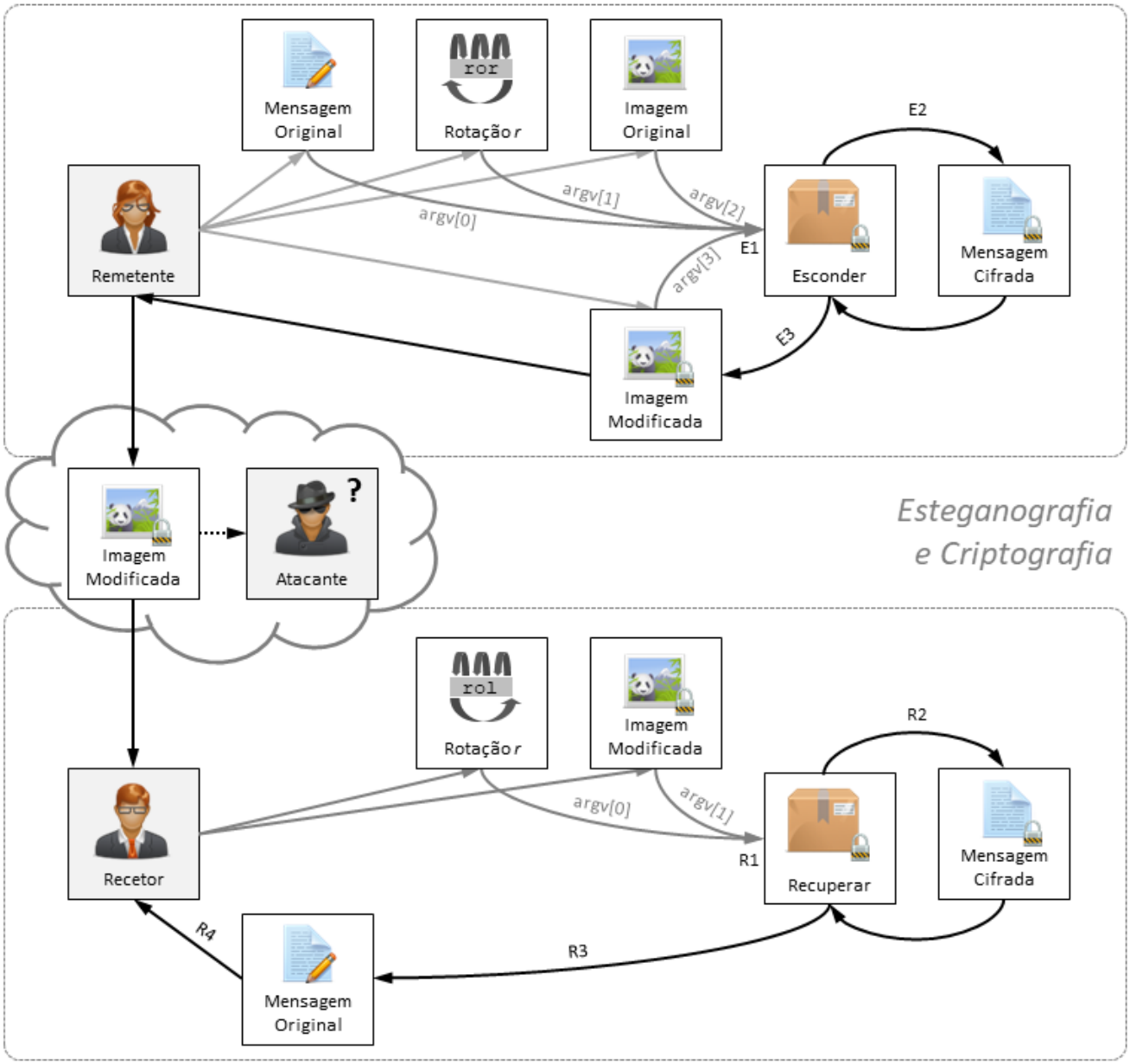

Figura 1: Exemplo de fluxo de execução do uso da esteganografia e criptografia.

Após a *Mensagem Cifrada* ser ocultada na *Imagem Original*, a *Imagem Modificada* resultante pode ser enviada através de um meio de comunicação qualquer para um *Recetor*. Caso a *Imagem Modificada* seja intercetada por alguém (p. ex., um *Atacante*), este indivíduo apenas verá uma imagem normal, acima de qualquer suspeita. Pode-se assumir que o *Atacante* descartará a *Imagem Modificada* que ele intercetou caso ele não conheça esteganografia, não esteja a verificar se esta técnica foi aplicada nas imagens obtidas, ou até mesmo esteja a verificar a técnica, mas não saiba que a mensagem de texto escondida está cifrada.

A seguir, o *Recetor* recebe o ficheiro da *Imagem Modificada*. Ele então executa outro programa, chamado *Recuperar*, o qual extrairá desta imagem a *Mensagem Cifrada* de texto e a decifrará para obter a *Mensagem Original* de texto em claro. O programa *Recuperar* recebe apenas dois argumentos, nesta ordem: o parâmetro de *Rotação r* e o caminho para o ficheiro da *Imagem Modificada* (em BMP), e realiza os seguintes passos:

R1. Verifica se recebeu a quantidade correta de argumentos e imprime uma mensagem de erro em caso negativo.

R2. Aplica o processo inverso da esteganografia na *Imagem Modificada* para obter a *Mensagem Cifrada*;

R3. Aplica o processo inverso do algoritmo de criptografia (i.e., desta vez será uma rotação para a esquerda de *r* vezes, `rol r`) nos bytes da *Mensagem Cifrada* para recuperar a *Mensagem Original* de texto.

R4. Imprime na saída do terminal (*stdout*) a *Mensagem Original* de texto recuperada.

Nas próximas duas secções são detalhados os formatos de ficheiros que serão utilizados neste projeto: o formato de texto TXT e o formato de imagem *bitmap* (BMP). A seguir, também serão apresentadas uma motivação sobre o uso das operações lógicas de rotação em criptografia (Secção 2.4), uma introdução sobre como esconder uma mensagem de texto num ficheiro de imagem (Secção 2.5), a apresentação dos ficheiros de apoio ao projeto fornecidos juntamente com este enunciado (Secção 2.6) e exemplos de como utilizar o `hexdump` para visualizar os bytes de um ficheiro de imagem (Secção 2.7).

2.2 O formato de texto TXT

O formato de texto TXT armazena os caracteres de uma mensagem. Cada caractere da mensagem é codificado em um byte (8 bits) conforme os valores definidos pela tabela da convenção ASCII. Como os valores em decimal da tabela ASCII vão do 0 ao 127, estes poderiam ser armazenados com apenas 7 bits cada. Porém para facilitar o trabalho, vamos assumir 1 byte por caractere, onde o bit mais significativo é sempre o 0 (zero) e os sete bits menos significativos são os bits normais do valor em ASCII.

Este formato não possui cabeçalho, pelo que o primeiro byte do ficheiro corresponde já ao byte do primeiro caractere. O término do ficheiro é marcado pelo caractere ASCII de valor 0 (`0x00`, *zero*), pois este corresponde ao *null*. Note que é possível também usar caracteres especiais que representam uma nova linha (`0x0A`, *new line*), uma tabulação (`0x09`, *horizontal tab*), etc.

Há diversas formas de se criar um ficheiro de texto TXT, seja através da linha de comandos da consola, de um editor de texto em modo texto na consola (p. ex., o *nano* no Linux), ou um editor de texto em modo gráfico (p. ex., o *gedit* no Linux ou o *Notepad* no Windows). Por exemplo, para escrever e ler um ficheiro de texto a partir da linha de comandos da consola em Linux, pode-se utilizar os seguintes comandos:

```
$ echo "Uma mensagem de texto" > mensagem.txt
$ cat mensagem.txt
Uma mensagem de texto
```

2.3 O formato de imagem *bitmap* (BMP)

Os ficheiros *bitmap* (BMP) são ficheiros de imagens onde o valor exato de cada píxel (*px*) da imagem é apresentado explicitamente. Os ficheiros no formato BMP são divididos em duas partes: o **cabeçalho** e a secção dos **píxeis**. Estas duas partes são apresentadas na Figura 2.

O cabeçalho pode seguir uma de diversas especificações existentes e pode ter um tamanho variável. Porém, para este trabalho, será utilizada a especificação ARGB32 (que será explicada a seguir) e as únicas informações que serão importantes de ler do cabeçalho serão o tamanho do ficheiro (*size*), o qual está indicado como **SSSS** na Figura 2, e o deslocamento (*offset*), o qual está indicado como **OOOO** na mesma figura. O *size* é um número de 4 bytes de tamanho que começa no 3º byte do ficheiro e indica o tamanho total do ficheiro BMP em bytes. O *offset* também é um número de 4 bytes de tamanho, porém começa no 11º byte do ficheiro e indica exatamente em qual byte a secção dos píxeis inicia.

A secção dos píxeis começa a partir desta posição de deslocamento (*offset*) e é composta por conjuntos também de 4 bytes (i.e., 32 bits) por píxel, representados por **BGRA** na Figura 2. Cada conjunto **BGRA** contém 1 byte para a intensidade (um número entre 0 e 255) da cor azul (**B**), 1 byte para o verde (**G**), 1 byte para o vermelho (**R**) e 1 byte para a transparência (**A**, que representa o chamado canal *alfa* e vai de 0 para totalmente transparente a 255 para totalmente opaco).

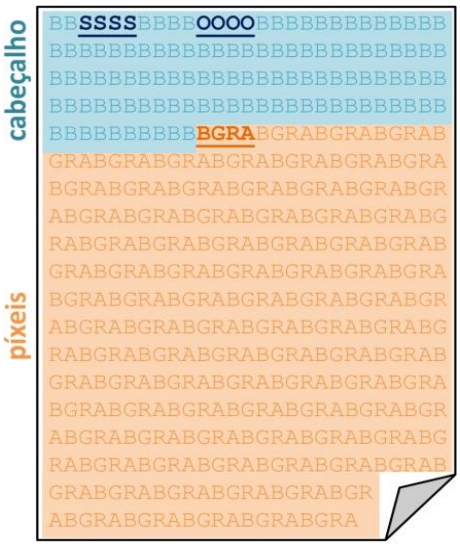

Figura 2: Estrutura de um ficheiro BMP

Esta especificação é chamada ARGB32 e em arquiteturas *little-endian* utilizam esta sequência **BGRA** para cada píxel. Portanto, um píxel composto pelos bytes **0x0000FFFF** é um píxel vermelho puro sem transparência, um **0x00FF00FF** é um verde puro, um **0xFF0000FF** é um azul puro, um **0x00FFFFFF** é um píxel amarelo e assim por diante. Qualquer píxel do tipo **0x??????00** é um píxel transparente. Neste trabalho, vamos sempre utilizar o valor **0xFF** (i.e., 255) para a transparência (ou seja, o byte A de cada píxel). O fim da secção dos píxeis (e consequentemente do ficheiro BMP) corresponde ao byte na posição indicada pelo número **SSSS** da Figura 2.

2.4 As operações lógicas de rotação em criptografia

As operações lógicas de rotação binária (p. ex., `ror` e `rol`), apesar de simples, são extremamente úteis a diversos algoritmos de criptografia pelas mais diversas razões, nomeadamente:

- por estarem disponíveis na grande maioria dos processadores modernos;
- por serem extremamente eficientes, pois são executadas num único ciclo de processamento;
- pelo seu tempo de execução ser independente do tamanho dos dados utilizados, prevenindo ataques de temporização (i.e., *timing attacks*);
- por serem capazes, quando combinadas a outras operações lógicas (p. ex., NAND), de criar difusão. Um algoritmo de criptografia possui uma boa difusão se ele gerar, mesmo para mensagens de texto muito semelhantes (p. ex., "*exemplo*" e "*Exemplo*"), cifras muito diferentes (p. ex., "*4ksW6Yy*" e "*xR19e7L*").

O método de criptografia que será utilizado neste projeto consiste apenas numa rotação binária, que será aplicada nos caracteres (i.e., bytes) da mensagem de texto antes de escondê-la na imagem. Este método permite ocultar o significado da mensagem de maneira simples. Porém, note que o mesmo é pouco seguro, pois possui pouca difusão e basta um atacante adivinhar o lado e o número de rotações utilizados para cifrar cada caractere da mensagem original de texto que ele conseguirá facilmente realizar o processo inverso para decifrar a mensagem.

2.5 Como esconder uma mensagem de texto numa imagem?

Para o caso específico deste projeto, vamos esconder uma mensagem cifrada de texto (que originalmente está presente num ficheiro TXT) numa imagem no formato BMP. Primeiramente, obtém-se o conteúdo binário da mensagem de texto original, onde cada caractere encontra-se codificado em 1 byte, conforme a tabela ASCII. A seguir, deve-se cifrar cada caractere (i.e., byte) da mensagem original. É também preciso ler o ficheiro da imagem original onde vai se esconder a mensagem cifrada e detetar o início da secção dos píxeis nesta imagem (ver a Secção 2.3).

Após estes passos, para esconder uma mensagem de texto numa imagem, coloca-se cada bit da mensagem de texto no bit menos significativo (LSB, do inglês *Least Significant Bit*) de cada byte das cores dos píxeis de índice par da imagem (B no índice 0 e R no índice 2). Os bytes de índice ímpar dos píxeis da imagem (i.e., G no índice 1 e A no índice 4) não serão utilizados para guardar

os bits da mensagem. Isso torna o algoritmo de esteganografia deste projeto menos usual (os algoritmos tradicionais utilizam todos os bytes de cor, incluindo o G) e evita variações de transparência (A) na imagem modificada que poderiam chamar a atenção caso a imagem modificada fosse intercetada.

Como apenas o bit menos significativo de cada byte de cor de índice par é alterado, o resultado desta transformação será uma imagem visualmente idêntica à imagem original, porém com uma mensagem cifrada escondida nela. Devido ao facto de utilizarmos em cada píxel apenas 2 bytes de cor (o B e o R), será possível guardar 2 bits da mensagem cifrada por cada píxel da imagem. A título de curiosidade, este modelo implica que uma imagem com `N×N`*px* possa guardar até $(N^2 \times 2)/8$ caracteres duma mensagem.

A ordem com que os bits da mensagem de texto são inseridos nos bytes do píxeis de cor de índice par da imagem é muito importante. Esta ordem deve ser seguida tanto para esconder a mensagem na imagem, como para recuperá-la. Mais especificamente, deve-se respeitar esta ordem:

- O **1º** bit do **1º** caracter da mensagem será guardado no LSB do 1º byte (índice 0, B) do 1º píxel da imagem.
- Nada será guardado no 2º byte (índice 1, G) do 1º píxel da imagem, pois este tem índice ímpar.
- O **2º** bit do **1º** caracter da mensagem será guardado no LSB do 3º byte (índice 2, R) do 1º píxel da imagem.
- Nada será guardado no 4º byte (índice 3, A) do 1º píxel, pois este é o byte do canal *alfa*.
- O **3º** bit do **1º** caracter da mensagem será guardado no LSB do 1º byte (índice 0, B) do 2º píxel da imagem.
- ...
- O **8º** bit do **1º** caracter da mensagem será guardado no LSB do 3º byte (índice 2, R) do 4º píxel da imagem.
- Nada será guardado no 4º byte (índice 3, A) do 4º píxel, pois este é o byte do canal *alfa*.
- O **1º** bit do **2º** caracter da mensagem será guardado no LSB do 1º byte (índice 0, B) do 5º píxel da imagem.
- Novamente, nada será guardado no 2º byte (índice 1, G) do 5º píxel.
- O **2º** bit do **2º** caracter da mensagem será guardado no LSB do 3º byte (índice 0, B) do 5º píxel da imagem.
- E assim por diante conforme a Figura 3.

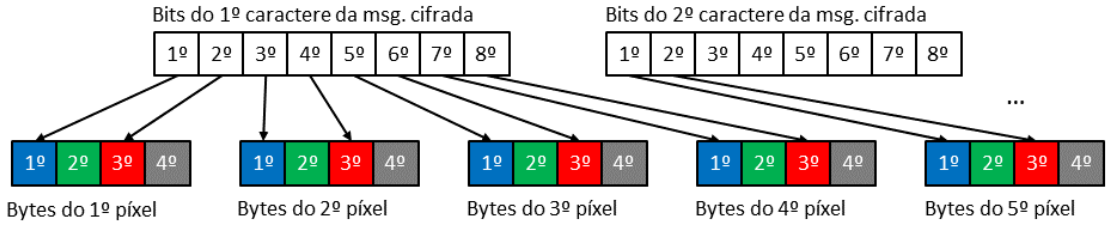

Figura 3: Diagrama para exemplificar a distribuição dos bits da mensagem cifrada pelos bytes de índica par dos píxels da imagem.

2.6 Ficheiros de apoio ao projeto

Junto ao enunciado, é disponibilizado um ficheiro (***apoio_projeto.tar.gz***) com diversos ficheiros BMP e uma biblioteca de funções para auxiliar a implementação do projeto:

- **Ficheiro básico:** O ficheiro ***básico.bmp*** serve para auxiliar a compreensão da estrutura dos ficheiros BMP e auxiliar nos estágios iniciais da implementação, pois possuem uma complexidade baixa ao nível de cores. Este ficheiro possui um tamanho de `10×10`*px*, onde todos os píxeis possuem a cor *Lapis Lazuli* sem transparência (i.e., `0xFF336699`), pois esta facilita a visualização das componentes das cores dos seus píxeis.
- **Ficheiros de teste:** Estes ficheiros servem para auxiliar a implementação dos programas e realizar testes para as fases da implementação (as quais serão descritas na Secção 3.1). São disponibilizados três ficheiros, cada um com um tamanho de `150×150`*px*, os quais possuem uma complexidade elevada de cores. O ficheiro ***fcul.bmp*** auxiliará a implementação do programa Esconder, pois ele é um ficheiro original que não possui nenhuma mensagem cifrada inserida nele. Os ficheiros ***fcul_mod_0_em_claro.bmp*** e ***fcul_mod_5_cifrada.bmp*** auxiliarão a implementação do programa Recuperar pois eles são ficheiros que já possuem mensagens ocultas neles. O primeiro possui uma mensagem em claro (i.e., com o valor zero para a Rotação *r*) oculta através do uso de esteganografia, enquanto o segundo possui uma mensagem cifrada (com o valor 5 para a Rotação *r*).

- **Biblioteca de funções:** Esta biblioteca contém algumas funções úteis para o desenvolvimento do projeto. As principais funções implementadas nela incluem a leitura de uma mensagem original (armazenada num ficheiro de texto em TXT) para a memória (`readMessageFile`), a leitura de um ficheiro de imagem BMP para a memória (`readImageFile`) e a escrita de um ficheiro de imagem BMP a partir de um buffer (i.e., espaço) na memória (`writeImageFile`). Algumas funções secundárias também são disponibilizadas, nomeadamente uma para finalizar a execução do programa (`terminate`) e outra para escrever uma *string* no terminal de saída (`printStrLn`).

Para além destes ficheiros, os alunos podem futuramente criar ou utilizar qualquer outra imagem BMP com o código implementado, desde que a especificação utilizada nos ficheiros BMP seja a ARGB32 (conforme mencionado na Secção 2.3). O código pode ainda ser modificado para aceitar outras especificações, mas isto é um trabalho fora do escopo deste projeto.

2.7 Visualização de uma imagem BMP com o comando `hexdump`

Pode-se utilizar o comando `hexdump` para visualizar o conteúdo binário (em hexadecimal, octal, decimal ou ASCII) de um ficheiro de imagem BMP (p. ex., o ficheiro ***basico.bmp*** descrito na Secção 2.3). A seguir são apresentados alguns exemplos de comandos `hexdump` que serão úteis para o desenvolvimento deste projeto:

1. Imprimir todo o conteúdo de um ficheiro, byte a byte, onde cada byte é apresentado com 2 dígitos hexadecimais na ordem em que os bytes são armazenados em arquiteturas *little-endian*:

```
   $ hexdump -v -e '1/1 "%02X "' basico.bmp
42 4D 1A 02 00 00 00 00 00 00 8A 00 00 00 7C 00 00 00 0A 00 00 00 0A 00 00 00 01
00 20 00 03 00 00 00 90 01 00 00 23 2E 00 00 23 2E 00 00 00 00 00 00 00 00 00 00
00 00 FF 00 00 FF 00 00 FF 00 00 00 00 00 00 FF 42 47 52 73 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 99 66 33 FF 99 66 33 FF 99 66 33 FF …
```

2. Imprimir o tamanho (em bytes) do ficheiro BMP, o qual está guardado no seu cabeçalho. Esta informação está armazenada nos 4 bytes a partir do 3º byte do ficheiro. Para visualizá-la, basta saltar 2 bytes (com o argumento `-s`) e definir o tamanho da leitura para 4 bytes (com o argumento `-n`):

```
$ hexdump -v -s 2 -n 4 -e '1/4 "Tamanho (em hexadecimal): %08X\n"' basico.bmp
Tamanho (em hexadecimal): 0000021A

$ hexdump -v -s 2 -n 4 -e '1/4 "Tamanho (em decimal): %d\n"' basico.bmp
Tamanho (em decimal): 538
```

3. Pode-se validar o tamanho obtido no comando anterior através do comando `du`:

```
$ du -b basico.bmp
538
```

4. Imprimir o deslocamento (*offset*) para o início da secção dos píxeis da imagem (i.e., o fim do cabeçalho) no ficheiro BMP, o qual também está presente no cabeçalho do mesmo. Esta informação está armazenada nos 4 bytes a partir do 11º byte do ficheiro. Para visualizá-la, basta saltar 10 bytes (i.e., `-s 10`) e definir o tamanho da leitura para 4 bytes (i.e., `-n 4`):

```
$ hexdump -v -s 10 -n 4 -e '1/4 "Offset (em hexadecimal): %08X\n"' basico.bmp
Offset (em hexadecimal): 0000008ª

$ hexdump -v -s 10 -n 4 -e '1/4 "Offset (em decimal): %d\n"' basico.bmp
Offset (em decimal): 138
```

5. Imprimir somente os píxeis do ficheiro de imagem BMP, a cada 4 bytes (i.e., 32 bits) no formato **ARGB**. Note que agora o salto (`-s`) deve ser igual ao tamanho do *offset* obtido com o comando anterior e que a ordem dos dados apresentados difere da ordem em que eles estão armazenados (ver o primeiro exemplo do comando `hexdump`):

```
$ hexdump -v -s 138 -e '1/4 "%02X "' basico.bmp
FF336699 FF336699 FF336699 …
```

3 Implementação

3.1 Fases da implementação

A entrega do projeto consiste numa única data (ver Secção 3.1), porém recomenda-se que a sua implementação seja feita em duas fases separadas:

1) **Recuperar:** A primeira fase consiste em começar por implementar o programa ***fcXXXXX_Recuperar.asm***, onde XXXXX é o número do aluno. Esta é a parte mais simples e permitirá aos alunos contextualizarem-se melhor com o formato BMP e com algumas funcionalidades que serão úteis também na fase seguinte. Como descrito na Secção 2.6, é vos dado dois ficheiros BMP (***fcul_mod0_em_claro.bmp*** **e** ***fcul_mod_5_cifrada.bmp***) com mensagens ocultas, as quais o vosso programa deverá conseguir recuperar com Rotação *r* igual a 0 e 5 respetivamente.
2) **Esconder:** A segunda fase consiste em implementar o programa ***fcXXXXX_Esconder.asm***, pois este é mais complexo e o seu resultado deverá funcionar corretamente com o programa *Recuperar* criado na fase anterior. Neste caso, os alunos devem utilizar a outra imagem BMP de teste (***fcul.bmp***), pois esta é considerada uma imagem original, sem nenhuma mensagem oculta nela.

3.2 Detalhes de compilação e execução

- A biblioteca de funções auxiliares ***Biblioteca.asm*** fornecida (ver Secção 2.6) deve ser compilada com o seguinte comando:
  ```
  $ nasm -F dwarf -f elf64 Biblioteca.asm
  ```

- O programa ***fcXXXXX_Esconder.asm*** será compilado, ligado à biblioteca de funções auxiliares fornecida e executado através dos seguintes comandos:
  ```
  $ nasm -F dwarf -f elf64 fcXXXXX_Esconder.asm
  $ ld fcXXXXX_Esconder.o Biblioteca.o -o Esconder
  $ ./Esconder mensagem.txt 3 fcul.bmp fcul_mod.bmp
  ```

- De modo semelhante, o programa ***fcXXXXX_Recuperar.asm*** será compilado, ligado à biblioteca de funções auxiliares fornecida e executado através dos seguintes comandos:
  ```
  $ nasm -F dwarf -f elf64 fcXXXXX_Recuperar.asm
  $ ld fcXXXXX_Recuperar.o Biblioteca.o -o Recuperar
  $ ./Recuperar 3 fcul_mod.bmp
  ```

A execução destes programas considera o mesmo exemplo duma Rotação *r* arbitrária (p. ex., 3) para cifrar e decifrar a mensagem original. O que muda é apenas o lado em que esta rotação é feita (i.e., para a direita no Esconder e para a esquerda no Recuperar). Para além disso, utilizar o valor 0 (zero) para a Rotação *r* implica que a mensagem será escondida em claro (i.e., sem ser cifrada) na imagem original, como é o caso do ficheiro auxiliar ***fcul_mod_0_em_claro.bmp***. Por último, o caractere '`3`' utilizado como exemplo de argumento nos comandos acima é na realidade um caractere ASCII e deverá ser convertido para um número decimal antes de ser armazenado como o valor da Rotação *r* na memória.

Note que os comandos apresentados nesta secção assumem que todos os ficheiros de código *Assembly*, os ficheiros de texto das mensagens, os ficheiros das imagens e a biblioteca fornecida devem estar todos na mesma diretoria para que estes funcionem corretamente. Pode haver casos (p. ex., *debug* com o SASM) em que é necessário passar o caminho absoluto de cada ficheiro (p. ex., os caminhos começados com o caractere "`/`" como em "`/home/aluno-di/ASC/projeto/mensagem.txt`") como argumentos.

3.3 Detalhes e sugestões para a implementação

Muitas das funcionalidades a serem implementadas já foram tema das aulas de ASC ou são abordadas nas referências bibliográficas da disciplina. Seguem alguns exemplos:

- A validação e obtenção dos argumentos passados a um programa: Permitirá receber, através da linha de comandos, a Rotação *r* e o caminho para os ficheiros a serem utilizados pelos programas. Este tema foi abordado nos Exercícios 3 do [ASC9a] e 2 do [ASC9b];
- A conversão de caracteres ASCII para dígitos numéricos: Permitirá converter a Rotação *r* de um caractere ASCII passado como argumento ao programa para o seu valor numérico. Este tema foi abordado no Exercício 5 da ficha de exercícios [ASC5];
- A escrita de mensagens na saída do terminal (*stdout*): Permitirá escrever a mensagem resultante do programa Recuperar. Este tema foi abordado nos guiões [ASC9a] e [ASC9b];
- A leitura de ficheiros: Permitirá ler a mensagem do ficheiro TXT a ser ocultada na imagem, assim como ler as imagens BMP (p. ex., a imagem original no programa Esconder e a imagem modificada no programa Recuperar). Este tema é abordado pela Secção 13.8.2 do livro [Jor20]. Note que na parte final da secção "*Read from file*" do código deste exemplo, o registo *rax* guarda na verdade a quantidade de bytes que foram lidos do ficheiro.
- A escrita de ficheiros: Permitirá escrever a imagem BMP modificada pelo programa Esconder. Este tema é abordado pela Secção 13.8.1 do livro [Jor20];
- Encontrar o tamanho (*size*) e o deslocamento (*offset*) nos ficheiros BMP: Permitirá saber onde começa e onde termina a secção dos píxeis nestes ficheiros. Ver a Secção 2.3.
- Percorrer a memória byte-a-byte: Permitirá obter separadamente cada byte da mensagem ou cada byte dos píxeis da imagem BMP. Este tema foi abordado em diversos exercícios e programas com exemplos de ciclos com endereçamento indexado.

4 Entrega do trabalho

4.1 Data da entrega

O trabalho é **individual** e deverá ser entregue na seguinte data:

- Até às **23:59** do dia **23/01/2022**.

4.2 Formato da entrega

A entrega será feita através da página da disciplina no Moodle, antes do limite do período de entrega, em local assinalado para tal. Caso os alunos não sigam exatamente as regras especificadas a seguir, serão penalizados na nota:

- Os dois ficheiros *Assembly* devem ser gravados exatamente com os nomes ***fcXXXXX_Recuperar.asm*** e ***fcXXXXX_Esconder.asm***, onde XXXXX é o número do aluno.
- Os alunos devem incluir, no início de cada ficheiro *Assembly* entregue, uma linha de comentário com o seu número de aluno (exatamente "; fcXXXXX").
- Os dois ficheiros *Assembly* devem ser colocados em um ficheiro comprimido ZIP ou TAR.GZ com o número de aluno como o nome do ficheiro (exatamente ***fcXXXXX.zip*** ou ***fcXXXXX.tar.gz***).

- O ficheiro comprimido (***fcXXXXX.zip*** ou ***fcXXXXX.tar.gz***) deve conter **apenas** os dois ficheiros *Assembly* implementados pelo aluno e **não** deve incluir o ficheiro ***Biblioteca.asm*** fornecido.

4.3 Critérios de avaliação

- **[1 valor]** Compilação do código e ligação das bibliotecas a funcionar sem erros. Serão utilizados os comandos apresentados na Secção 3.2 e outros semelhantes.
- **[4 valores]** O código realiza a validação do número de argumentos passados ao programa e utiliza corretamente cada argumento na ordem especificada nas Secções 2.1 e 3.2.
- **[5 + 5 valores]** Funcionamento correto dos programas *Esconder* e *Recuperar* conforme especificado neste enunciado. Serão executados os comandos apresentados na Secção 3.2 com os ficheiros de teste, para além de outros ficheiros BMP e outras mensagens de texto.
- **[3 valores]** Conformidade estrita com o formato de entrega descrito na Secção 4.2.
- **[1 valor]** Organização dos códigos em *Assembly*. Aspetos que serão valorizados incluem respeitar as convenções de chamada de funções *System V*, aplicar boas práticas de uso da memória, evitar referências à memória desnecessárias, etc.
- **[1 valor]** Documentação dos códigos em *Assembly*. Aspectos que serão valorizados incluem comentar o funcionamento de trechos de códigos mais complexos e não-triviais.

4.4 Plágio

Não é permitido aos alunos partilharem códigos com soluções, ainda que parciais, de nenhuma parte do projeto com outros alunos (nem através do Fórum da disciplina, nem por qualquer outro meio). Além disso, todos os códigos serão testados por um verificador de plágio. Caso alguma irregularidade seja encontrada, os projetos de todos os alunos envolvidos serão anulados e o caso será reportado aos órgãos responsáveis na Ciências@ULisboa.

Por fim, é responsabilidade de cada aluno garantir que a sua *home*, as suas diretorias e os seus ficheiros de código estão protegidos contra a leitura de outras pessoas que não o utilizador dono dos mesmos. Por exemplo, se os ficheiros estiverem gravados na sua área de aluno nos servidores da Ciências@ULisboa, então todos os itens mencionados anteriormente devem ter as permissões de acesso 700 (ver o último item da Secção 1.3 do Guião [ASC1]).

5 Bibliografia

[ASC1] M. Calha, V. Cogo., S. Signorello. (2021). Guião de Laboratório – Linux. Disponível online em https://moodle.ciencias.ulisboa.pt/mod/resource/view.php?id=153677.

[ASC5] M. Calha, V. Cogo., S. Signorello. (2021). Ficha de exercícios – x86: Operações aritméticas e lógicas. Disponível online em https://moodle.ciencias.ulisboa.pt/mod/resource/view.php?id=167402.

[ASC9a] M. Calha, V. Cogo., S. Signorello. (2021). Ficha de exercícios – x86: Pilha. Disponível online em https://moodle.ciencias.ulisboa.pt/mod/resource/view.php?id=169087.

[ASC9b] M. Calha, V. Cogo., S. Signorello. (2021). Guião de laboratório – x86: Pilha. Disponível online em https://moodle.ciencias.ulisboa.pt/mod/resource/view.php?id=169228.

[Jor20] Ed Jorgensen. (2020). x86-64 Assembly Language Programming with Ubuntu. Disponível online em http://www.egr.unlv.edu/~ed/x86.html.