Análise do código/Relatório - Laboratórios de Informática II

Dinis Peixoto A75353 Marcelo Miranda A74817
José Sousa A74678

May 31, 2015

1 Código gerado

```
0x08048400 <contar_segs+0>:     push %ebp
0x08048401 <contar_segs+1>:     mov %esp,%ebp
0x08048403 <contar_segs+3>:     push %edi
0x08048404 <contar_segs+4>:     push %esi
0x08048405 <contar_segs+5>:     push %ebx
0x08048406 <contar_segs+6>:     sub $0xc,%esp
0x08048409 <contar_segs+9>:     xor %edi,%edi
0x0804840b <contar_segs+11>:    xor %ecx,%ecx
0x0804840d <contar_segs+13>:    xor %edx,%edx
0x0804840f <contar_segs+15>:    cmpb $0x0,0x2720(%ebp)
0x08048416 <contar_segs+22>:    mov 0x2724(%ebp),%eax
0x0804841c <contar_segs+28>:    movl $0x0,-0x10(%ebp)
0x08048423 <contar_segs+35>:    movl $0x0,-0x14(%ebp)
0x0804842a <contar_segs+42>:    je 0x8048486<contar_segs+134>
0x0804842c <contar_segs+44>:    lea -0x1(%eax),%ecx
0x0804842f <contar_segs+47>:    movl $0x1,-0x10(%ebp)
0x08048436 <contar_segs+54>:    mov 0x2718(%ebp),%eax
0x0804843c <contar_segs+60>:    test %eax,%eax
0x0804843e <contar_segs+62>:    jle 0x804847b<contar_segs+123>
0x08048440 <contar_segs+64>:    mov %eax,%esi
0x08048442 <contar_segs+66>:    lea 0x0(,%edx,4),%eax
0x08048449 <contar_segs+73>:    add %edx,%eax
0x0804844b <contar_segs+75>:    lea (%ecx,%ecx,4),%ebx
0x0804844e <contar_segs+78>:    mov %eax,-0x18(%ebp)
0x08048451 <contar_segs+81>:    lea 0x0(%esi),%esi
0x08048454 <contar_segs+84>:    lea (%ebx,%ebx,4),%eax
0x08048457 <contar_segs+87>:    lea 0x8(%ebp,%eax,4),%eax
0x0804845b <contar_segs+91>:    sub $0xc,%esp
0x0804845e <contar_segs+94>:    movsbl (%edi,%eax,1),%eax
0x08048462 <contar_segs+98>:    push %eax
0x08048463 <contar_segs+99>:    call 0x80483e4<e_seg>
0x08048468 <contar_segs+104>:   add $0x10,%esp
0x0804846b <contar_segs+107>:   test %al,%al
0x0804846d <contar_segs+109>:   je 0x8048472<contar_segs+114>
```

```
0x0804846f <contar_segs+111>:    incl -0x14(%ebp)
0x08048472 <contar_segs+114>:    add -0x10(%ebp),%edi
0x08048475 <contar_segs+117>:    add -0x18(%ebp),%ebx
0x08048478 <contar_segs+120>:    dec %esi
0x08048479 <contar_segs+121>:    jne 0x8048454<contar_segs+84>
0x0804847b <contar_segs+123>:    mov -0x14(%ebp),%eax
0x0804847e <contar_segs+126>:    lea -0xc(%ebp),%esp
0x08048481 <contar_segs+129>:    pop %ebx
0x08048482 <contar_segs+130>:    pop %esi
0x08048483 <contar_segs+131>:    pop %edi
0x08048484 <contar_segs+132>:    leave
0x08048485 <contar_segs+133>:    ret
0x08048486 <contar_segs+134>:    lea -0x1(%eax),%edi
0x08048489 <contar_segs+137>:    mov $0x1,%edx
0x0804848e <contar_segs+142>:    mov 0x271c(%ebp),%eax
0x08048494 <contar_segs+148>:    jmp 0x804843c<contar_segs+60>
```

2 Tabela de Registos

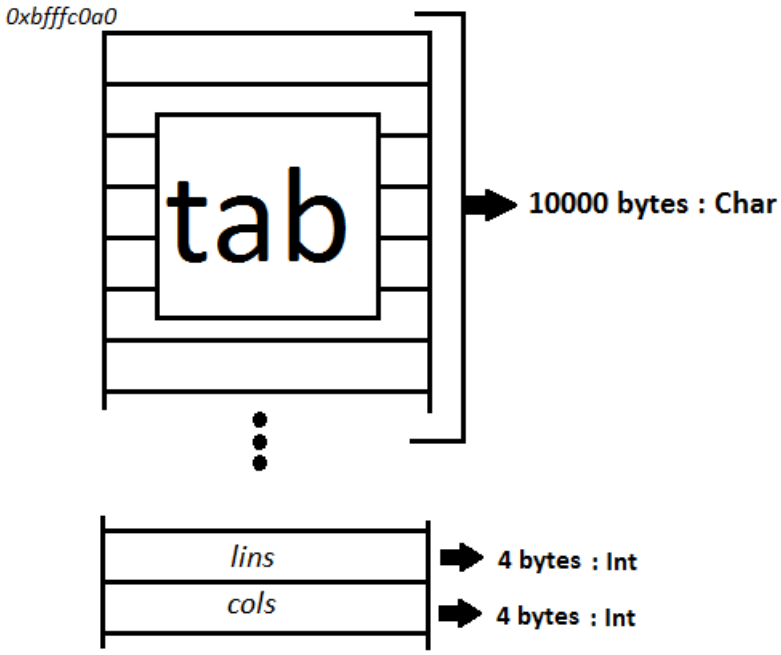
Registo	Conteúdo
%eax	*num*;
%eax	*tam*;
%eax	variação da base utilizada para calcular a linha;
%eax	posição do tabuleiro;
%eax	resultado *e_seg*;
%ecx	y;
%edx	dy;
%ebx	base para o cálculo da linha;
%esi	tam;
%edi	x;

3 Variável *tab*

Utilizando o *gdb* concluimos que a variável *tab* se localiza no endereço *0xbfffc0a0*. Esta corresponde a uma estrutura que contém uma matriz de caracteres(*char*) e dois inteiros(*int*). Tendo em conta que um *char* ocupa 1 byte, e que um *int* ocupa 4 bytes, como se trata de uma matriz de 100 por

100, então esta matriz ocupa $100 * 100 = 10000$ bytes, que adicionados aos $4 * 2 = 8$ bytes dos inteiros, dá um total de $10000 + 8 = 10008$ bytes.

3.1 Representação na Memória



4 Análise de código

Para inicializar a função, e tendo em conta os registos que vão ser utilizados na mesma, é necessário salvaguardar o antigo conteúdo destes, para que possam futuramente ser recuperados. O registo %ebp é responsável pela base de uma **stack frame**, neste caso da função chamadora *main*, como estámos a inicializar a função *contar_segs*, este vai ter de ser alterado para construir uma nova stack frame, e vai ficar a apontar para o mesmo sítio que o registo %esp (topo da stack frame da função *main*).

```
0x08048400 <contar_segs+0>:   push %ebp        # Salvaguarda o registo;
0x08048401 <contar_segs+1>:   mov %esp,%ebp    # Iguala o %ebp ao %esp;
0x08048403 <contar_segs+3>:   push %edi        # Salvaguarda o registo;
0x08048404 <contar_segs+4>:   push %esi        # Salvaguarda o registo;
0x08048405 <contar_segs+5>:   push %ebx        # Salvaguarda o registo;
```

Após a inicialização da função surge a instrução **sub** $0xc,%esp, que aumenta 12 bytes na

stack, reservando assim memória para 3 variáveis. Isto é sucecido de três instruções idênticas **xor**, que tal como instrução anterior fazem uma subtração entre dois registos, esta instrução é aplicada, nos três casos, a dois registos iguais, atribuindo-lhes assim valor 0, $x = 0$, $y = 0$ e $dy = 0$.

```
0x08048406 <contar_segs+6>:   sub $0xc,%esp          # reserva memória para 3 variáveis;
0x08048409 <contar_segs+9>:   xor %edi,%edi          # x = 0;
0x0804840b <contar_segs+11>:  xor %ecx,%ecx          # y = 0;
0x0804840d <contar_segs+13>:  xor %edx,%edx          # dy = 0;
```

Segue-se uma instrução que compara (fazendo para isso uma subtração entre o valor da variável em memória **0x2720(%ebp)** e 0), isto é representado pela condição **if(lin)** na função. É atribuída, pela primeira vez, uma variável ao registo %eax, a variável *num*, além disto é também atribuído às variáveis *dx* e *count* o valor 0. A comparação feita anteriormente será decisiva agora, pois segue-se uma instrução **je**. Como a variável *lin* só pode tomar o valor 0 ou 1, caso esta tenha valor 0 o resultado do **cmpb** será também 0 e o salto irá realizar-se para o endereço **0x8048486<contar_segs+134>**, caso contrário prossegue.

```
0x0804840f <contar_segs+15>:  cmpb $0x0,0x2720(%ebp)          # compara o valor da variavel lin;
0x08048416 <contar_segs+22>:  mov 0x2724(%ebp),%eax           # %eax = num;
0x0804841c <contar_segs+28>:  movl $0x0,-0x10(%ebp)           # dx = 0;
0x08048423 <contar_segs+35>:  movl $0x0,-0x14(%ebp)           # count = 0;
0x0804842a <contar_segs+42>:  je 0x8048486<contar_segs+134>   # if (lin == 0);
```

A instrução que se segue **lea** realiza uma adição entre o valor da variável contida no registo %eax, que no momento é *num* e -1 e guardando o resultado desta operação no registo %ecx onde está contida a variável *y*, como vimos anteriormente. São também executadas duas mais instruções, **movl** $0x1,-0x10(%ebp) e **mov** 0x2718(%ebp),%eax, com o objectivo de atribuir 1 à variável *dx* e guardar o conteúdo da memória **0x2718(%ebp)**, o valor de *t.lins*, no registo %eax, que passa assim a conter a variável *tam*. As seguintes instruções **test** %eax,%eax e **jle** 0x804847b<contar_segs+123> testam se o programa vai entrar no ciclo for ou não, caso o test retorne 0 o salto condicional vai-se realizar. O **mov** %eax,%esi guarda a variável *tam* no resgisto %esi.

```
0x0804842c <contar_segs+44>:  lea -0x1(%eax),%ecx          # y = num − 1;
0x0804842f <contar_segs+47>:  movl $0x1,-0x10(%ebp)        #dx = 1;
0x08048436 <contar_segs+54>:  mov 0x2718(%ebp),%eax        #%eax = tam = t.lins;
0x0804843c <contar_segs+60>:  test %eax,%eax               # Testa se (tam! = 0);
0x0804843e <contar_segs+62>:  jle 0x804847b<contar_segs+123> # Salto condicional: (tam > 0);
0x08048440 <contar_segs+64>:  mov %eax,%esi                # $esi = tam;
```

As intruções que se seguem dizem respeito à **indexação da matriz**. **Funcionamento da matriz em C** Os elementos consecutivos das linhas da matriz estão contiguos na memória, ou seja, o primeiro elemento da segunda linha encontra-se imediatamente a seguir ao último elemento da primeira linha. Tendo em conta que a matriz é $100x100$ se quisermos aceder a matriz[3][2] teremos de aceder à posição $3 * 100 + 2 = 302$.

Em Assembly

Antes da execução do ciclo for o programa começa por calcular uma base, partindo do valor de y (linha a que queremos aceder), que mais tarde vai ser usada para calcular a respectiva

posição na matriz(verificamos isto na instrução **lea** (%ecx,%ecx,4),%ebx), e da variação (0 ou 5) dessa mesmo base(**lea** 0x0(,%edx,4),%eax e **add** %edx,%eax. Dependendo do valor da base calculada, vamos chegar a uma linha da matriz diferente. Dependendo do valor da base, vamos chegar a uma linha da matriz diferente.
Exemplos:

Linha 0 Base 0;

Linha 1 Base 5;

Linha 2 Base 10;

```
0x08048442 <contar_segs+66>:    lea 0x0(,%edx,4),%eax          # Variação da
linha no ciclo for;
0x08048449 <contar_segs+73>:    add %edx,%eax                  # Variação da
linha no ciclo for;
0x0804844b <contar_segs+75>:    lea (%ecx,%ecx,4),%ebx         # Determina uma
base para a linha;
0x0804844e <contar_segs+78>:    mov %eax,-0x18(%ebp)           # Guarda variação
da linha na memória;
0x08048451 <contar_segs+81>:    lea 0x0(%esi),%esi             # NOP
```

Já dentro do ciclo for, são executados alguns cálculos auxiliares, **lea**(%ebx,%ebx,4),%eax e **lea** 0x8(%ebp,%eax,4),%eax, para determinar a posição da matriz correspondente à linha desejada, somando-lhe depois a coluna pretendida **movsbl** (%edi,%eax,1),%eax. O conteúdo da posição da matriz calculada anteriormente vai servir de argumento à função *e_segs*, daí a instrução **call** 0x80483e4<e_seg>.

```
0x08048454 <contar_segs+84>:    lea (%ebx,%ebx,4),%eax
0x08048457 <contar_segs+87>:    lea 0x8(%ebp,%eax,4),%eax
0x0804845b <contar_segs+91>:    sub $0xc,%esp
0x0804845e <contar_segs+94>:    movsbl (%edi,%eax,1),%eax
0x08048462 <contar_segs+98>:    push %eax
0x08048463 <contar_segs+99>:    call 0x80483e4<e_seg>
```

As instruções seguintes correspondem ao conteúdo do ciclo **for**. A primeira instrução **add** é responsável por baixar o stack pointer %esp, esta é seguida do **test** responsável por verificar se o *e_seg* retorna 0 ou 1, o resultado desta instrução é posteriormente utilizado pela instrução **je**, este salto condicional só se realiza caso a função auxiliar *e_seg* tenha retornado 0, e se isto acontecer salta apenas uma instrução, instrução esta (**incl**) responsável por incrementar 1 unidade na variável *count*, que só se realiza caso o valor de retorno de *e_seg* tenha sido 1. Segue-se então duas instruções semelhantes **add** -0x10(%ebp),%edi e **add** -0x18(%ebp),%ebx, responsáveis pelas operações $x+ = dx$ e $y = dy$ respectivamente, estas operações são necessárias para actualizar a linha/coluna em que o ciclo for vai atuar de seguida. A última tarefa a ser realizada no ciclo for, e como a variável i que percorre o ciclo foi removida ao compilar com a flag **-O2**, é a variável *tam* que percorre o ciclo e por isso a instrução **dec** %esi retira 1 unidade de cada vez que o ciclo é percorrido. Em vez do ciclo correr enquanto a condição $i < tam$ é realizada, este corre enquanto $tam > 0$, esta condição é assegurada pela instrução seguinte **jne** 0x8048454<contar_segs+84>, isto é, quando $tam! = 0$ o salto condicional dirige-nos novamente para o início do ciclo for.

```
0x08048468 <contar_segs+104>:   add $0x10,%esp
0x0804846b <contar_segs+107>:   test %al,%al                   # Verfica se
o e_seg retorna 0 ou 1;
0x0804846d <contar_segs+109>:   je 0x8048472<contar_segs+114>  # Salto condi-
cional se retornar 0;
0x0804846f <contar_segs+111>:   incl -0x14(%ebp)               # count++;
0x08048472 <contar_segs+114>:   add -0x10(%ebp),%edi           # x+ = dx;
```

0x08048475 <contar_segs+117>: add -0x18(%ebp),%ebx # $y+ = dy;$

0x08048478 <contar_segs+120>: dec %esi # $tam - -;$

0x08048479 <contar_segs+121>: jne 0x8048454<contar_segs+84> # *Salto condiciona se* $tam > 0$*;*

Por fim, as instruções que se seguem, excepto as quatro últimas, dizem respeito à finalização da função *e_segs*. Quando uma função tem valor de retorno é comum que este fique no registo %eax, é o que acontece na primeira instrução apresentada a seguir, a variável *count* que vai ser retorno da função é colocada neste mesmo registo. Posto isto, a instrução **lea** -0xc(%ebp),%esp é responsável por desalocar os 12 bytes de memória alocados na inicialização da função, bytes estes ocupados pelos argumentos. Os registos salvaguardados no ínicio pelos **push** são também recuperados através dos três **pop** que se seguem. A função está pronta para retornar para a função chamadora, após a execução das instruções **leave** e **ret**.
Estas quatro últimas instruções, dizem respeito à condição **if(lin)**, quando esta não é cumprida, ou seja, quando a variável $lin = 0$. Semelhante às instruções vistas anteriormente, as intruções **lea** -0x1(%eax),%edi , **mov** $0x1,%edx e **mov** 0x271c(%ebp),%eax correspondem à atribuição de $num - 1$ à variável x, à atribuição da constante 1 a dy e de *t.cols* a *tam*, respetivamente. NOTA: Como estas instruções são realizadas somente se o salto condicional se efetuar, o registo %eax contém a variável *tam*.

0x0804847b <contar_segs+123>: mov -0x14(%ebp),%eax # $\%eax = count;$

0x0804847e <contar_segs+126>: lea -0xc(%ebp),%esp # *Desalocar memória dos argumentos;*

0x08048481 <contar_segs+129>: pop %ebx # *Recupera o registo;*

0x08048482 <contar_segs+130>: pop %esi # *Recupera o registo;*

0x08048483 <contar_segs+131>: pop %edi # *Recupera o registo;*

0x08048484 <contar_segs+132>: leave

0x08048485 <contar_segs+133>: ret # *Retornar;*

0x08048486 <contar_segs+134>: lea -0x1(%eax),%edi # $x = num - 1;$

0x08048489 <contar_segs+137>: mov $0x1,%edx # $dy = 1;$

0x0804848e <contar_segs+142>: mov 0x271c(%ebp),%eax # $\%eax = tam = t.cols;$

0x08048494 <contar_segs+148>: jmp 0x804843c<contar_segs+60>

4.1 Código em C

```
int contar_segs(tabuleiro t, bool lin, int num) {
    int i;
    int x = 0;                                  # 0x08048409 <contar_segs+9>:   xor     %edi,%edi
    int y = 0;                                  # 0x0804840b <contar_segs+11>:  xor     %ecx,%ecx
    int dx = 0;                                 # 0x0804841c <contar_segs+28>:  movl    $0x0,-0x10(%ebp)
    int dy = 0;                                 # 0x0804840d <contar_segs+13>:  xor     %edx,%edx
    int tam = 0;
    int count = 0;                              # 0x08048423 <contar_segs+35>:  movl    $0x0,-0x14(%ebp)

    if(lin) {                                   # 0x0804842a <contar_segs+42>:  je      0x8048486 <contar_segs+134> (quando a condição não se cumpre)
        y = num - 1;                            # 0x0804842c <contar_segs+44>:  lea     -0x1(%eax),%ecx
        dx = 1;                                 # 0x0804842f <contar_segs+47>:  movl    $0x1,-0x10(%ebp)
        tam = t.lins;                           # 0x08048436 <contar_segs+54>:  mov     0x2718(%ebp),%eax

    } else {                                    # 0x0804842a <contar_segs+42>:  je      0x8048486 <contar_segs+134> (quando a condição se cumpre)
        x = num - 1;                            # 0x08048486 <contar_segs+134>: lea     -0x1(%eax),%edi
        dy = 1;                                 # 0x08048489 <contar_segs+137>: mov     $0x1,%edx
        tam = t.cols;                           # 0x0804848e <contar_segs+142>: mov     0x271c(%ebp),%eax
    }


    for(i = 0; i < tam; i++) {                  # 0x08048479 <contar_segs+121>: jne     0x8048454 <contar_segs+84> (condição que faz repetir o ciclo)
        if(e_seg(t.tab[y][x]))                  # 0x0804846d <contar_segs+109>: je      0x8048472 <contar_segs+114>
            count++;                            # 0x0804846f <contar_segs+111>: incl    -0x14(%ebp)
        x += dx;                                # 0x08048472 <contar_segs+114>: add     -0x10(%ebp),%edi
        y += dy;                                # 0x08048475 <contar_segs+117>: add     -0x18(%ebp),%ebx
    }

    return count;                               # 0x0804847b <contar_segs+123>: mov     -0x14(%ebp),%eax
}
```

4.2 Comando Resolver

A função *resolver* começa por aplicar todas as 4 estratégias(funções *strategy1*,*strategy2*,*strategy3* e *strategy4*) até que estas deixem de aplicar qualquer alteração ao tabuleiro considerado, isto é verficado pela condição $res > 0$ do ciclo *do while*. Posto isto, se o tabuleiro não tiver concluído e for válido $((checkSolution(tab) == 0 \;\&\&\; testTab(tab) == 1))$ entrámos na segunda parte desta função, responsável por completar o tabuleiro com "brute force", a função auxiliar responsável por esta parte é *bruteForce*.
Na função *bruteForce* começa por, através da função *probMatrix* calcula a matriz cujas posições têm a probabilidade de haver um fragmento ainda não descoberto. Posto isto, vai tentando colocar nas posições em que a probabilidade é mais alta o fragmento 'o' e tenta novamente resolver o tabuleiro. Caso isto não tenha dado certo , isto é, quando o tabuleiro deixa de ser válido ou está mal resolvido ou ainda se a proporção dos barcos não estiver correta, passa a colocar nas posições com probabilidade mais baixa, isto continuamente tentando assim resolver o tabuleiro. Se chegar ao ponto em que mesmo as posições com probabilidade 0 não permitem a resolução do tabuleiro, podemos concluir que o tabuleiro não apresenta uma solução válida.