1.

What is loop unrolling? How can it make your program more efficient? How can it make your program less efficient?

Loop unrolling is processing more than one iteration in just one iteration. The loop counter would increment by more than one each time, but the loop itself would already deal with the increments. It can make programs more efficient by pipelining different instructions because there is less dependence of instructions. It can make programs less efficient because if a loop is unrolled too many times, the registers may spill because more values need to be stored at once.

2.

What affects the data stored on the stack? In the context of the attack lab, what instructions should be paid careful attention to?

Procedure calls affect the data stored on the stack. Changes to %rsp and procedure calls should be paid careful attention to.

3.

The provided function func\_one takes as input two pointers, that are actually each individually pointing to the first element in a N by M array of integers.

int func\_one(char\* one, char\* two, int N, int M) {

int i, j, k;

int sum = 0;

char\* ptr1 = one;

char\* ptr2 = two;

int rowBytes = 4\*M;

for (k = 0; k < 4; k++) {

int shift = k << 3;

for (j = 0; j < M; j++) {

int intBytes = j\*4;

for (i = 0; i < N; i++) {

char one = \*(ptr1 + k + intBytes + i\*rowBytes);

int masked = one & 0xFF;

~~int shift = k << 3;~~

int shifted = masked << shift;

\*(ptr2 + k + intBytes + i\*rowBytes) = masked;

sum += shifted;

}

}

}

return sum;

}

In what ways can we optimize the above function?

Also reorder the loops to be I, j, then k to maximize spatial locality.

4.

The provided code below is an optimization of the previous problem. Fill in the blanks.

int func\_two(char\* one, char\* two, int N, int M) {

int i, j, k;

int sum = 0;

int temp = 0;

char\* ptr1 = one;

char\* ptr2 = two;

for (i = 0; i < N; i++) {

for (j = 0; j < M; j++) {

temp = (0xFF & \*ptr1);

sum += temp;

\*ptr2 = temp;

ptr1++;

ptr2++;

temp = (0xFF & \*ptr1);

sum += temp << 8;

\*ptr2 = temp;

ptr1++;

ptr2++;

temp = (0xFF & \*ptr1);

sum += temp << 16;

\*ptr2 = temp;

ptr1++;

ptr2++;

temp = (0xFF & \*ptr1);

sum += temp << 24;

\*ptr2 = temp;

ptr1++;

ptr2++;

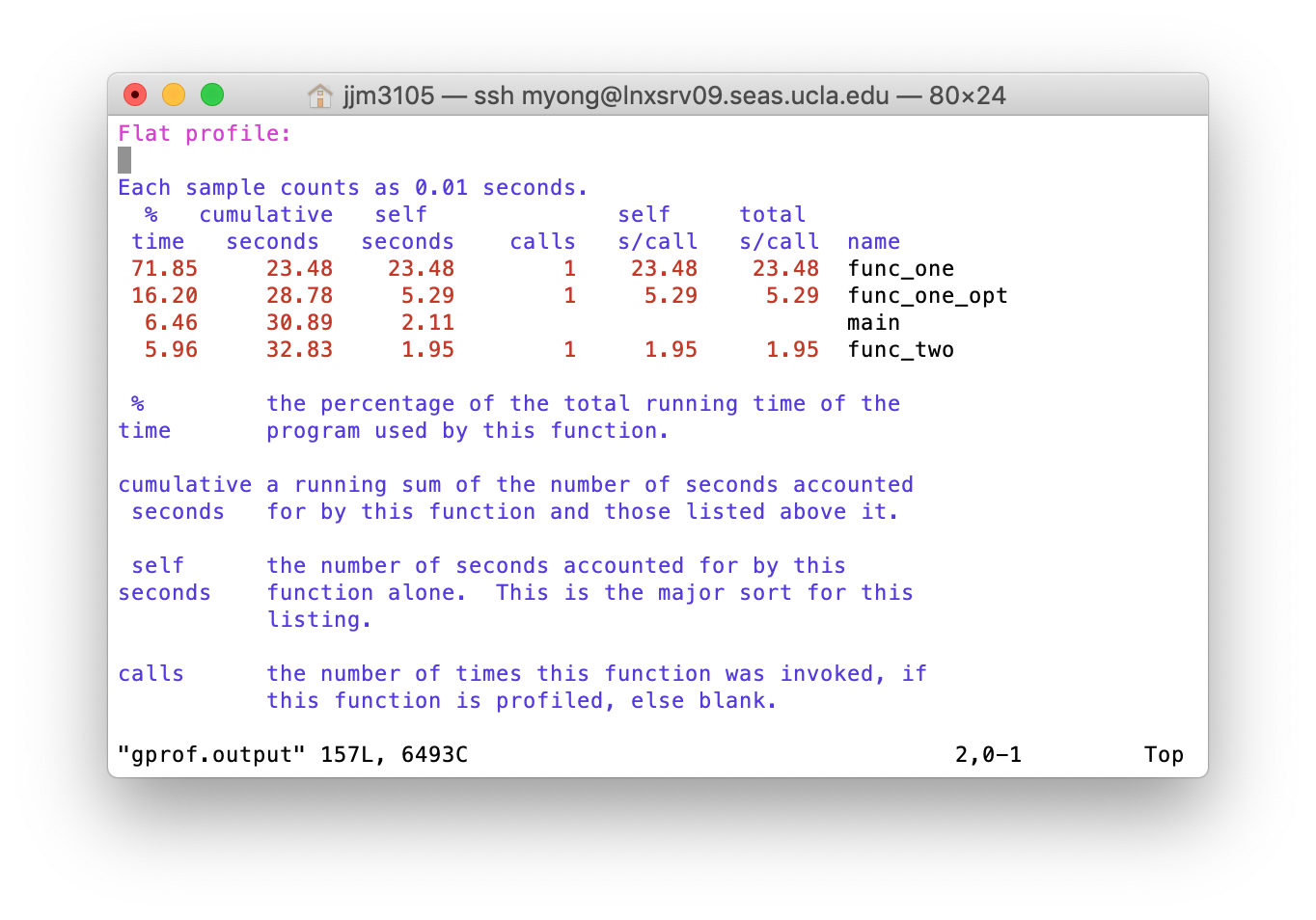
}

}

return sum;

}

FYI.



The above shows the running time of the functions discussed in problems 5 and 6. func\_one is the code from problem 5 as is, func\_one\_opt is one optimization of func\_one, and func\_two is the completed code from problem 6.

The results were generated using gprof