1.
What is loop unrolling? How can it make your program more efficient? How can it make your program less efficient?

Loop unrolling is technique that reduces the number of iterations of a loop. For example, let's say you have a for loop that iterates 100 times and prints "hello" once each time. You could transform it into a for loop that iterates 50 times prints "hello" twice each time. Loop unrolling makes your program more efficient, since you don't need to make the comparison every time. It also tells the CPU you can run instructions out of order or in parallel. Loop unrolling can make your program less efficient if you have a lot of temporary variables in a single iteration, requiring too many registers. It also makes your code less readable, and some compilers will do the optimization for you.

2.
What affects the data stored on the stack? In the context of the attack lab, what instructions should be paid careful attention to?

Any instructions performed on the stack pointer register %rsp should be monitored; this includes add instructions and sub instructions. pop and push instructions also affect the layout of the stack; these typically work (on a 64-bit machine) by either popping/pushing values off of/onto the stack.

3.
The following table gives the parameters for a different number of caches. For each cache, fill in the missing fields in the table.

- *m* is the number of physical address bits
- *C* is the cache size
- *B* is the block size
- *E* is the associativity
- *S* is the number of the cache sets
- *t* is the number of tag bits
- *s* is the number of set index bits
- *b* is the number of block offset bits

Cache	*m*	*C*	*B*	*E*	*S*	*t*	*s*	*b*
1	32	1024	4	4	64	24	6	2
2	32	2048	4	4	128	23	7	2
3	32	1024	8	1	128	22	7	3
4	32	1024	8	128	1	29	0	3
5	32	1024	8	8	16	25	4	3
6	32	1024	32	4	8	24	3	5

There are four relevant formulas:

$C = B * E * S$

$m = t + s + b$

$B = 2^b$

$S = 2^s$

4.
Assume the following:

- The memory is byte addressable.
- Memory accesses are to 1-byte words (not to 4-byte words).
- Addresses are 13 bits wide.
- The cache is two-way set associative (E = 2), with a 4-byte block size (B = 4) and eight sets (S = 8).

The contents of the cache are as follows, with all numbers given in hexadecimal notation.

	Line 0						Line 1					
Set Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	09	1	86	30	3F	10	00	0				
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0					0B	0				
3	06	0					32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0				
6	91	1	A0	B7	26	2D	F0	0				
7	46	0					DE	1	12	C0	88	37

Suppose a program running on a machine with such a cache references the 1 byte word at the address 0x0E34.

Recall the addresses are in the form {Tag}{Index}{Offset}
From the assumptions, we know that the lowest 2 bits are for offset and the next 3 bits for index.
0x0E34 is 01110001_101_00

What is the resulting of the following?

Cache block offset: 0x0
Cache set index: 0x5
Cache tag: 0x71
Cache hit? Yes
Cache byte returned: 0xB

5.
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;

      for (k = 0; k < 4; k++) {
         for (j = 0; j < M; j++) {
            for (i = 0; i < N; i++) {

               char one = *(ptr1 + k + j*4 + i*4*M);
               int masked = one & 0xFF;

               int shift = k << 3;
               int shifted = masked << shift;

               *(ptr2 + k + j*4 + i*4*M) = masked;
               sum += shifted;
            }
         }
      }

      return sum;
}
```

In what ways can we optimize the above function?

There are several ways to improve upon this function.

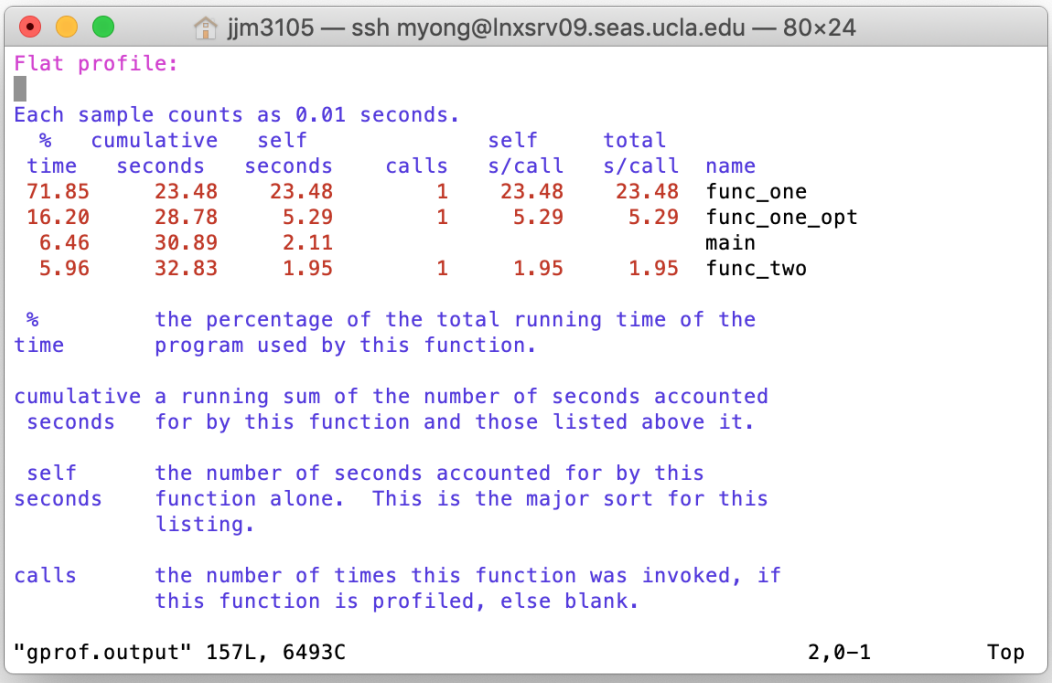
- Following the principles of **spatial locality**, we can switch the order of the for loops to be in i, j, then k order.
- We can pull out the multiplication “j*4” into the middle loop, the “shift” variable into the outer loop, and 4*M out of all the loops.

The function essentially copies the array “one” into “two”, and returns the sum of all the elements in array “one” while its at it. Note that it is not essential to understand what exactly the function does, to optimize it. Knowing what it does, however, allows us to restructure the code.

6.
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.