CS232 - Digital Logic Design & Computer Architecture Lab

Siddharth Mahesh Patil
21d070071

February 25, 2023

1 Question 1

For this question, the command I have used to disassemble the provided executable files is:

```
$ objdump -C -d -M intel <executable file> > <output file>
```

The final flag we get is "**CS230{R3v3rse_Engine3ring_is_easy!!}**"
To read through the code, firstly, I made a simple cpp program which accepts an integer and outputs it from a function and then I tried to understand the control flow and used this knowledge to solve this question. I also used the x86 cheat-sheet to figure out what each function did and which registers it used/modified.

1.1 Part A

For Part A, the input x is valid if $x > 5000$ and the secret key I am getting when I input x as 5000 is 836766038 and the sub-string of the flag is "CS230{". We get this constraint from the following code segment:

```
1376:        mov    DWORD PTR [rbp-0x24],edi
1379:        cmp    DWORD PTR [rbp-0x24],0x1387
1380:        jle    1428 <part_a(int)+0xbf>
```

⋮

```
1428:         lea    rsi,[rip+0x1cf4]       # 3123
<std::__detail::_S_invalid_state_id+0x7b>
```

The code corresponding to 1379 and 1380 checks whether the input x is less than or equal to 4999 (0x1387), and if it is, then the code jumps to the invalid state (corresponding to 1428).

1.2 Part B

For Part B, the inputs a, b, c are valid if $a^2 + b^2 = c^2$ and the value of the secret key, when I enter $a = 3, b = 4, c = 5$, is 1699638299 and the sub-string of the flag is "is_easy!!}". We get the constraints from the following code snippet:

```
1376:        mov    DWORD PTR [rbp-0x24],edi
1379:        mov    DWORD PTR [rbp-0x28],esi
137c:        mov    DWORD PTR [rbp-0x2c],edx
137f:        mov    eax,DWORD PTR [rbp-0x24]
1382:        imul   eax,eax
1385:        mov    edx,eax
1387:        mov    eax,DWORD PTR [rbp-0x28]
138a:        imul   eax,eax
138d:        add    edx,eax
138f:        mov    eax,DWORD PTR [rbp-0x2c]
1392:        imul   eax,eax
1395:        cmp    edx,eax
1397:        jne    143f <part_b(int, int, int)+0xd6>
```

⋮

```
143f:        lea    rsi,[rip+0x1cdd]         # 3123
<std::__detail::_S_invalid_state_id+0x7b>
```

In the above code snippet, the "imul" operations are calculating the squares of the inputs a, b, c and the line corresponding to 138d is adding a^2 and b^2 and then, at the lines corresponding to 1395 and 1397, it is comparing $a^2 + b^2$ with c^2 and jumping to the invalid state if they are not equal.

1.3 Part C

For Part C, the input string *str* is valid if its length is either 7 or 9 and the string is a palindrome (i.e., the string when reversed is the same as the original string). The secret number I am getting is 772873549 when *str* = "*aaaaaaa*" and the sub-string of the flag is "R3v3rse_Engine3ring_". In this case, first the code checks the length and then checks whether the string is a palindrome or not. Code snippet to check the size:

```
1416:         mov    QWORD PTR [rbp-0x38],rdi
141a:         mov    rax,QWORD PTR [rbp-0x38]
141e:         mov    rdi,rax
1421:         call   11f0 <strlen@plt>
1426:         mov    DWORD PTR [rbp-0x24],eax
1429:         cmp    DWORD PTR [rbp-0x24],0x6
142d:         jle    143f <part_c(char*)+0x36>
142f:         cmp    DWORD PTR [rbp-0x24],0xa
1433:         jg     143f <part_c(char*)+0x36>
1435:         mov    eax,DWORD PTR [rbp-0x24]
1438:         and    eax,0x1
143b:         test   eax,eax
143d:         jne    1457 <part_c(char*)+0x4e>
143f:         lea    rsi,[rip+0x1cc1]        # 3107
<std::__detail::_S_invalid_state_id+0x5f>
```

In the above code segment, on 1421, the code is calculating the length of the string and then on 1429, 142d, its comparing the length with 6 and jumping to the invalid state if the length is less than or equal to 6. Then, on 142f and 1433, its comparing the length with 10 and jumping to an invalid state if length is greater than 10. On 1438-143d, the code is checking whether the length is even or odd, and if it is odd, then it is jumping to some part of the code, but if it is even then the control flow takes it to the invalid state.
The following is the code segment to check if the string is a palindrome:

```
1462:         call   11b0 <operator new[](unsigned long)@plt>
1467:         mov    QWORD PTR [rbp-0x20],rax
146b:         mov    DWORD PTR [rbp-0x2c],0x0
1472:         mov    eax,DWORD PTR [rbp-0x2c]
1475:         cmp    eax,DWORD PTR [rbp-0x24]
1478:         jge    14a5 <part_c(char*)+0x9c>
147a:         mov    eax,DWORD PTR [rbp-0x24]
147d:         sub    eax,0x1
1480:         sub    eax,DWORD PTR [rbp-0x2c]
1483:         movsxd rdx,eax
1486:         mov    rax,QWORD PTR [rbp-0x38]
148a:         add    rax,rdx
148d:         mov    edx,DWORD PTR [rbp-0x2c]
1490:         movsxd rcx,edx
1493:         mov    rdx,QWORD PTR [rbp-0x20]
1497:         add    rdx,rcx
149a:         movzx  eax,BYTE PTR [rax]
```

```
149d:          mov     BYTE PTR [rdx],al
149f:          add     DWORD PTR [rbp-0x2c],0x1
14a3:          jmp     1472 <part_c(char*)+0x69>
```

⋮

```
14d1:          call    12b0 <strcmp@plt>
14d6:          test    eax,eax
14d8:          jne     1580 <part_c(char*)+0x177>
```

⋮

```
1580:          lea     rsi,[rip+0x1b80]        # 3107
<std::__detail::_S_invalid_state_id+0x5f>
```

In the above code snippet, 1462 is declaring a new string and then the statements from 1472-14a3 form a loop to store the reverse of the input string *str* in the newly created string. After the reverse of the input string is calculated and stored, 14d1-14d8 compare the original string and its reverse and if they are different (not palindrome), then the program jumps to the invalid state.

2 Question 2 (+ BONUS)

To solve this question, originally I had used the Extended Euclidean Algorithm iteratively for 32 bit integers. But for the bonus part we needed to do these calculations for 64 bit numbers and the Extended Euclidean Algorithm uses division, multiplication and modulus operations, all of which are hard and inefficient for 64 bit numbers in MIPS because the default register size is 32 bits. Therefore, I have used Binary Euclidean Algorithm to solve this question.

2.1 I/O (Parsers)

Since MIPS is a 32 bit ISA, we cannot, directly, accept 64 bit integers. To tackle this problem, I have created two parsers, one which converts a string to an int (for Input), and the other converts an int to a string (for Output). For Input, my code first accepts the inputted number as a string, and then checks every character of the string and converts it to int to store it in the designated registers. This requires the multiplication of a 64 bit number with 10, but this can be done in a simpler way than two 64 bit integers because we already know the value of one of the numbers. For Output, my code converts a 64 bit integer to a string by dividing it by 10 taking its modulus and then storing these numbers as characters in the string. Again, this can be done, comparatively, easily because we already know one of the numbers.

2.2 Binary Euclidean Algorithm

Let's say that we have two co-prime integers, A and M. Then there are three possibilities, either both are odd, A is even and M is odd or M is even and A is odd.Lets look at all of these possibilities.

2.2.1 A is odd and M is even

In this case we can solve the same problem for A and $M/2$ and use its solution to get our solution. Eg. Let M be even and we want x and y which satisfy $A.x - M.y = 1$ and $x < M$. Then, we can solve for A and $M/2$ and get their solutions x' and y'.

$$(A).x' - \frac{M}{2}.y' = 1$$

If y' is even:

$$\implies x = x',\ y = \frac{y'}{2}$$

If y' is odd:

$$\implies x = x' + \frac{M}{2},\ y = \frac{A + y'}{2}$$

This is because we can add A to y' and $M/2$ to x' and the equality will still hold true and new $y' + A$ will be even and then we get the case when A was even.

2.2.2 M is odd and A is even

Similar to the case above, in this case, we can solve a sub-problem for $A/2$ and M and then get our solution from this solution. The manipulation is exactly the same as above after replacing A and M, therefore I haven't shown it here.

2.2.3 Both Odd

If both are odd, then we can solve the same problem on $min(A, M)$ and $|M - A|$ and get our answer from the answer of this sub-problem. Eg. Let $A > M$ and we want x and y which satisfy $A.x - M.y = 1$ and $x < M$. Then, we can solve for $A - M$ and M and get their solutions x' and y'.

$$(A - M).x' - M.y' = 1$$

$$\implies A.x' - M.(x' + y') = 1$$

$$\implies x = x',\ y = x' + y'$$

We can also do a similar manipulation if $A < M$.
Note: $|A - M|$ is an even integer, so we can apply the cases before this one to this sub-problem.

Using the above cases, we can recursively solve our problem in $O(log(n))$ time because after each iteration, either one of A or M is getting halved. The base cases needed are:

- If $A = 1 \implies x = 1,\ y = 0$
- If $M = 1 \implies x = 1,\ y = A - 1$

Note: This approach only requires addition, subtraction and left/right shifts (multiplication/division by 2) which are much easier to implement and more efficient than division and modulus required in the Extended Euclidean Algorithm.

3 Question 3

In this question, we had to implement merge-sort on an array with the constraints that the running time is $O(nlog(n))$ and the auxiliary space used is $O(1)$. A normal merge-sort takes $O(nlog(n))$ and $O(n)$ auxiliary space (for merging). We were also given the constraint that the elements in the array are $0 \leq A[i] \leq 10000$ implying that the input is at most of 14 bit integers, but, we are storing them in 32 bits. Therefore, we can use the upper bits of the storage as a temporary storage required during the merge step of merge-sort.

Therefore, to solve this question, I have used iterative merge-sort where I am using the upper 16 bits of each array element as temporary storage for the merge step and after the merging is done, the code will copy the upper 16 bits to the lower 16 bits and then clear the upper 16 bits to make it ready for the next merge step. In an iterative merge-sort, we first merge sub-arrays of size 1, then of size 2 and then of size 4 and so on and so forth until the size of the sub-arrays to be merged becomes more than the size of the entire array, when we stop. Since this is an iterative procedure, there is no need to maintain a stack of function calls, therefore, the auxiliary space used is $O(1)$ (constant number of variables/literals used).

Note: The question didn't mention in which order we had to sort the array, so I have sorted the array in descending order.

4 Question 4

In this question, we had to implement memory allocation for a matrix and matrix multiplication in x86.

4.1 Memory Allocation

I decided to store the matrix in the heap because of its flexibility when it comes to its max capacity. I have used the sys_brk command in x86 to do the memory allocation. sys_brk allocates memory by increasing the value of the program break point and this break point will be the starting address of the newly allocated data. Therefore, first, to find this address, I have called sys_brk with address as 0 which will give us the current program break address and then called sys_brk again with address set to the previous break point + number of bytes to allocate, this moves the program break to the address we have passed to it (if it is possible). *Note:* To call sys_brk, we have to set rax to 12 (code for sys_brk) and store the address we are sending as a parameter in rdi and then use syscall. The resulting address (updated program break) will be store in rax.

4.2 Matrix Multiplication

For matrix multiplication, I first wrote a simple code for matrix multiplication for ijk, copied it and then pasted it for all the other variants and modified the codes slightly according to the requirements (order of loops).

4.2.1 Optimization

The code I had written previously, although correct, wasn't very efficient, so I did some optimizations (where possible). The optimizations I have used are:

1. I have pre-computed the increments in the addresses of the array pointers, so that the code will only multiply by 8 once and then all the following changes in the addresses will be caused by addition or subtraction.

2. I have tried to store minimum amount of data on the stack because accessing elements from the stack is more time consuming compared to accessing data from the registers r0-r15. Therefore, I have used all the registers (r0-r15) and used the stack only when it was needed.

3. Whenever I have used the stack to store variables, I am using those variables in the outermost loop to minimise the number of times they are accessed.

4. Also, I haven't used cmp commands because whenever we do an addition or subtraction, the ZF flag is modified according to the result, and, we can use this instead of explicit cmp statements.

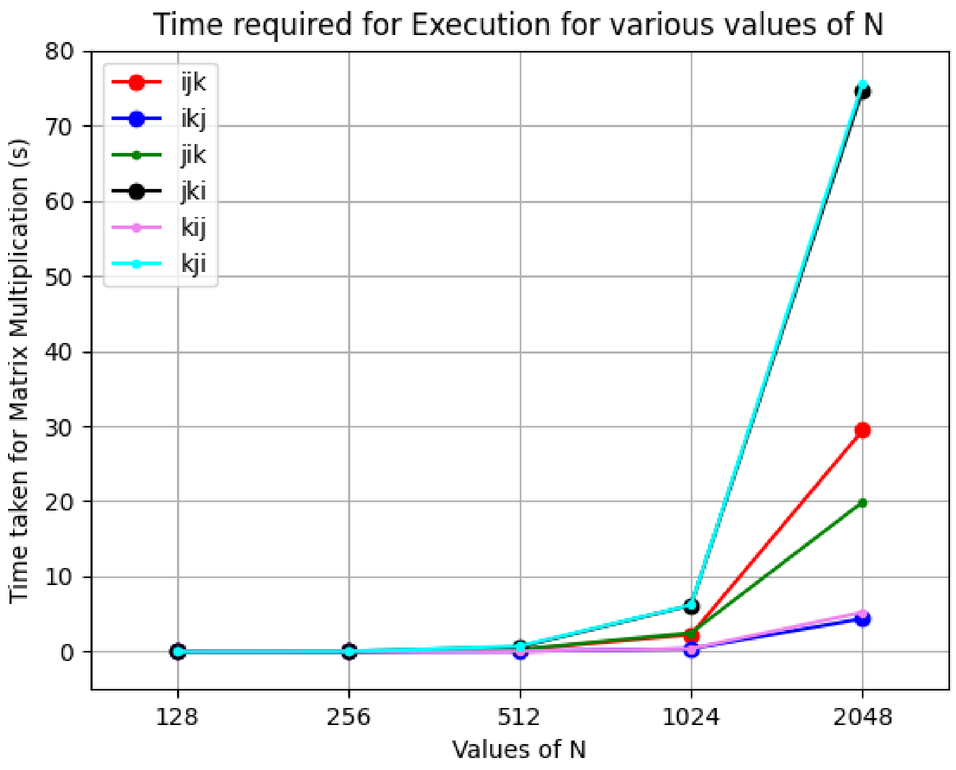
4.3 Data

IMPORTANT : The data (number of cycles) I was collecting kept varying considerably (over 10% difference sometimes) every-time I ran the code, so I am not very comfortable

with calling this data “conclusive” or “correct”, but, since the question demands it, I have listed the data from one of the data collection sessions (all of the data belongs to the same session, i.e., I didn’t run the code multiple times to find a faster time or a time which matches the behaviour of that program for other input sizes).
TSC frequency = 2688MHz

	N=128	N=256	N=512	N=1024	N=2048
ijk	4431131	25994078	664872185	6064710930	79102318853
ikj	3767858	23426509	129881016	1031282457	11769487176
jik	9298717	65373953	771459054	6680616998	53451912845
jki	12519457	105033562	1658110561	16577724269	200876815938
kij	2088340	17488742	158213938	1071198477	14067499945
kji	13558706	99632781	1955469438	16681849635	203073137102

Table 1: Cycles taken for various N for all matrix multiplication variations


Time required for Execution for various values of N
Time taken for Matrix Multiplication (s)
Values of N
ijk
ikj
jik
jki
kij
kji