EECS 444 Homework 2 Part 3

Yida Liu

October 24, 2019

1 Decode Assembly 1

1.1 Original Program

From inference on paper, we obtain the following code for the given assembly language.

```c
#include <stdio.h>

int main(int argc, char* argv, char* envp) {
    int a = 3, b = 5, c = 0;
    int d = (a * b) - (a / 2);
    c = d;
    printf("%d", c);
    return 0;
}
```

1.2 Decode Assembly

1.2.1 Initialize variables

```
; start of program
mov     dword ptr [esp+1Ch], 3
mov     dword ptr [esp+18h], 5
mov     dword ptr [esp+14h], 0
```

Initialize variables a, b, c to be used with values, 3, 5, 0, accordingly.

1.2.2 Arithmetic Operations

Operation 1: Multiplication

```
mov     eax, [esp+1Ch]
imul    eax, [esp+18h]
mov     edx, eax            ; edx = a * b
```

Save the result of a ∗ b to intermediate variable edx.

Operation 2: Integer Division

```
mov     eax, [esp+1Ch]
mov     ecx, eax
shr     ecx, 1Fh
add     eax, ecx
sar     eax, 1              ; eax = a / 2
```

This is common technique used for signed integer division. The code load a into eax and do right shift to obtain the right-most "sign" bit of eax. If eax is negative, add eax, ecx will add 1 to eax, otherwise 0. At last, a arithmetic right-shift is performed to do the divde by 2.

Therefore, we have eax to be a / 2.

Opertaion 3: Subtraction

```
sub     edx, eax          ; edx = edx - eax
mov     eax, edx          ; eax = edx
mov     [esp+14h], eax    ; c = eax
```

Perform the subtraction from previous steps, we have c = (a ∗ b) − (a / 2).

1.3 Printing

```
mov     eax, [esp+14h]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d"
call    _printf
; end of program
```

This basically prints c.

1.4 Summary

The output of the program is 14.

2 Decode Assembly 2

2.1 Original Program

The follow code shows the decoded assembly program in C.

```c
#include <stdio.h>

int main(int argc, char* argv, char* envp) {
    int array[8] = {0xC, 0xF, 0xDD, 3, 0x1B0, 0x36, 0x10, 0x43};
    int var = 0, cnt = 0;

    while(cnt <= 7) {
        if (array[cnt] > var) {
            var = array[cnt];
        }
        cnt += 1;
    }

    printf("%d", var);
}
```

2.2 Decode Assembly

We follow a bottom-up apprach to decode the given assembly program, starting from nodes (smaller chunks of logics).

2.2.1 Start of Program

In the start of program, a bunch of numbers / values are stored in the relative location of esp, which is the stack pointer.

2.2.2 Main Control Logic: loc_40157F

```
cmp     dword ptr [esp+38h], 7
jle     short loc_401560
mov     eax, [esp+3Ch]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d"
call    _printf
```

This logic is to do [esp+38h] $\leq$ 7 and jump to inner logic blocks if true, while inner logic blocks has a backward path to loc_40157F. Later, after the condition evaluates to false, the value of [esp+3Ch] is printed as a 32-bit integer. We could also be sure that [esp+3Ch] is the desired value to be printed, and [esp+38h] is a counter. We denote them as **var** and **cnt**, respectively.

2.2.3 loc_401560

```
mov     eax, [esp+38h]
mov     eax, [esp+eax*4+18h]
cmp     eax, [esp+3Ch]
jle     short loc_40157A
mov     eax, [esp+38h]
mov     eax, [esp+eax*4+18h]
mov     [exp+3Ch], eax
```

This part compares the value at the location cnt*4 + 18h relative to esp. We could see from Section 2.2.1, the initial defined value starts at [esp+18h]. Therefore, the pre-defined value must follow some array structure.

Next, $[esp + cnt * 4 + 18h] \leq var$ is evaluated, and the logic jump to loc_40157A if True. Otherwise, var will be assigned value of [esp+cnt*4+18h].

2.2.4 loc_40157A

```
add     dword ptr [esp+38h], 1
```

The counter cnt is incremented by 1; the logic lead back to loc_40157F.

2.3 Summary

In summary of the aforementioned analysis of each chunk of logic, we think that this program compares and finds the maximum value from a predefined array, whose value is shown in the Section 2.2.1. The program also maintains a counter to ensure the array access is not out of bound. **The output of the program is 432, which is the value [esp+28h], 1B0h**

3 Decode Assembly 3

3.1 Original Program

The following code shows the decoded assembly program in C.

```
#include <stdio.h>

int main(int argc, char* argv, char* envp) {

    int cnt = 0x64;
    int array[3];

    while (cnt < 0x3E7) {

        array[2] = cnt / 100;

        array[1] = (array[2] * -0x64 + cnt) / 10;

        array[0] = cnt - cnt / 10 * 10;

        if (array[0] * array[0] * array[0] +
            array[1] * array[1] * array[1] +
            array[2] * array[2] * array[2] == cnt)
            printf("%d", cnt);
        cnt += 1;
    }

    return 0;
```

3.2 Decode Assembly

3.2.1 Outer Logic: loc_4015D6 and loc_4015D1

First, a counter-like variable **cnt** at [esp + 1Ch] is defined; In addition to that, we see that the esp offse by 20h, which indicates some allocated space, as indicated by **array** that has 3 element in [esp+10h], [esp+14h], and [esp+18h], repsectively.

The outer logic of the program is essentially a while loop that is condition on the variable cnt that requires cnt to be less than 0x3E7 inside the loop. At the end of each loop, cnt is incremented by 1.

3.2.2 Inner Logic: loc_40151B

The patter of the inner logic could be found that at the end of each line of C code, the value at register is assigned back to the stack location. There are three assignments and one in-place conditional statement (i.e. if-statement). Therefore, we discuss them separatly.

Assign 1: [esp+18h] This chunk of code is from loc_40151B to loc_401534.

The presence of arbitrary constant 0x51EB851F is a form of alert that some compiler optimization happens here. Since it is followed by sar, we consider this chunk of code as some form of division, where the divisor is not a power of 2. Here's our justification on how the compiler did the optimization. Consider

a/b on integer a and constant b, the compiler performs the following computation:

$$\begin{aligned} a/b &= \frac{a}{b} \\ &= \frac{a * (1 << m)}{(1 << m) * b} \\ &= (\frac{a * (1 << m)}{b}) >> m \end{aligned}$$

where m is a positive integer and $m \geq 31$. Let

$$c = \frac{1 << m}{b}$$

, then we have

$$a/b = a * c >> m$$

Here in this case, we have $c = \texttt{0x51EB851F}, m = 31 + 5 + 1 = 37$, we have,

$$\begin{aligned} b &= (1 << m)/c \\ &= (1 << 37)/\texttt{0x51EB851F} \\ &= 99.99999997962732 \approx 100 \end{aligned}$$

At last, we do the assignment array [3] = cnt / 100.

Assign 2: [esp+14h] This chunk of code is from loc_401538 to loc_40155B.

```
mov     eax, [esp+18h]
imul    edx, eax, −64h
mov     eax, [esp+1Ch]
lea     ecx, [edx+eax]
```

This chunk of code computes ecx = array[1] ∗ −64h + cnt.

Then, in the following chunk, we see the similar structure of the aforementioned division optimization trick. We have $c = \texttt{0x66666667}, m = 34$, and compute the value of divisor b as follows:

$$\begin{aligned} b &= (1 << m)/c \\ &= (1 << 34)/\texttt{0x66666667} \\ &= 9.99999999650754 \approx 10 \end{aligned}$$

This snippet translates to array [2] = (cnt − array[3]∗100) / 10.

Assign 3: [esp+10h] This chunk of code is from loc_40155F to loc_401583.

We see the same division code from Paragraph 3.2.2. We have ecx = cnt; edx = ecx / 10.

The next chunk of code computes eax = edx; eax = eax ∗ 4 + edx; eax = eax + eax, which is equivalent to eax = edx ∗ 10.

Then, ecx = ecx − eax, which is equivalent to ecx = ecx − ecx / 10 ∗ 10.

At last, the value is assigned back to [esp+10h], shown as array [0] = cnt − cnt / 10 ∗ 10

If-Statement This chunk of code is from loc_401587 to loc_4015CC.

Essentially, this chunk computes the cube of each values in the array and sums them up; A equality check is performed for check if the print statement is triggered, where the value of cnt is printed.

3.3 Summary

In summary, we analyzed the functionality of the program. The program essentially computes three values along the loop and when sum of the cube of the three values equals the loop counter cnt, the value of the counter is printed. **The output of the program is 153370371407.** We reorganize the output and corresponding values in Table 1, in which we see that the values in the array is in fact the digits that composes the number.

array[0]	array[1]	array[2]	cnt
3	5	1	153
0	7	3	370
1	7	3	371
7	0	4	407

Table 1: Problem 3 Output and Corresponding Values

4 Decode Compiled Binary

We spent numerous hours tried to decompose by interpreting the program line by line and the logic of the assembly code just does not make sense to me. I had to utilize some tools to assist me. Lucily, IDA provided such tool for decompiling the program into c-like pseudocode. With the assistance IDA, we are able to obtain the following code.

```c
#include <stdio.h>

int proc1(int *a1, int a2, int a3)
{
    int v4; // [esp+0h] [ebp-10h]
    int v5; // [esp+4h] [ebp-Ch]
    int v6; // [esp+8h] [ebp-8h]
    int i;  // [esp+Ch] [ebp-4h]

    v5 = 0;
    v4 = 0;
    for (i = 0; i < a2; ++i)
    {
        v6 = 1;
        while (v6 < a3)
        {
            while (!a1[v5])
                v5 = (v5 + 1) % a2;
            ++v6;
            v5 = (v5 + 1) % a2;
        }
        while (!a1[v5])
            v5 = (v5 + 1) % a2;
        v4 = a1[v5];
        a1[v5] = 0;
    }
    return v4;
}

int main(int argc, const char **argv, const char **envp)
{
    int v3;       // eax
    int v5[100]; // [esp+14h] [ebp-19Ch]
    int v6;       // [esp+1A4h] [ebp-Ch]
    int v7;       // [esp+1A8h] [ebp-8h]
    int i;        // [esp+1ACh] [ebp-4h]

    v7 = 7;
    v6 = 100;
    for (i = 0; i < v6; ++i)
        v5[i] = i + 1;
    v3 = proc1(v5, v6, v7);
    printf("%d\n", v3);
    return 0;
}
```