CS271 Final Project Report

Aaron Frost

Modularity

My compute procedure is modularized into three sub-procedures: decoy, encrypt, and decrypt.

The user chooses which sub-procedure is used when they pass in the last argument over the stack before calling compute; this argument should be a pointer to a double-word. This double-word, which we refer to as “dest”, is either 0, -1, or -2, corresponding to decoy, encrypt, and decrypt accordingly, see figure 1.

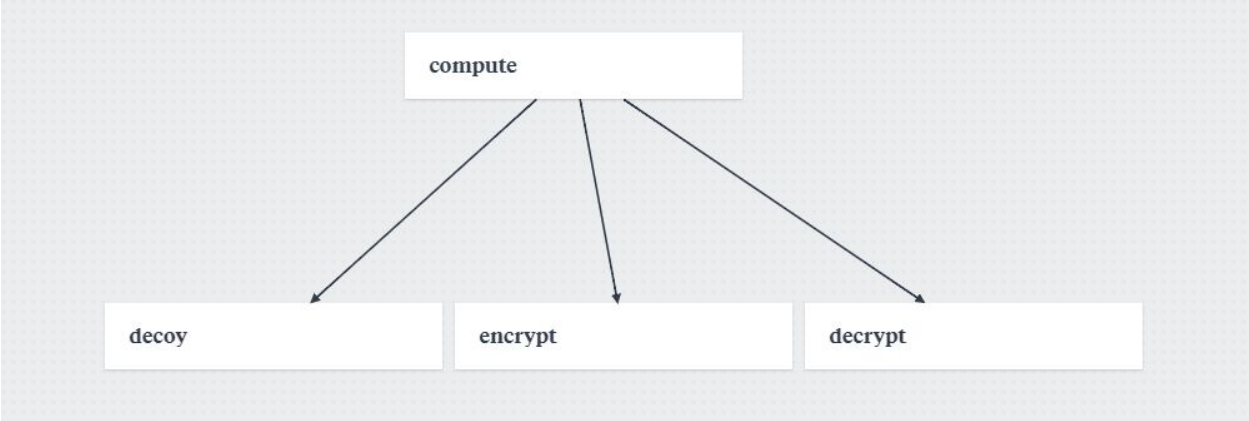

Fig. 1 - Modular Compute Function

Separating code into multiple procedures ensures that the elements of the program are nicer to read and easier to reuse or remove. It also entails pushing arguments using the system stack.

Challenges & Reconsiderations

Although I did not run out of general purpose registers, I was highly challenged in other areas of the assignment. I had a particularly hard time figuring out how to successfully pass variables over the stack, where it took some detailed observations of the stack memory changing to debug unexpected errors when trying to receive stack parameters. I found that using the watch, register, and memory debug windows were extremely helpful for locating offsets from the base pointer and otherwise just generally working with registers.

I used a scratch paper to formulate the pseudo-code for the decrypt and encrypt algorithms (see figure 2 & 3), which made it a lot easier to implement using assembly language. Even though I had a smooth time figuring out the general algorithms and structure of the program, I spent most of the time troubleshooting smaller problems, which often turned out to be trivial, such as

mismatching label names. Also, I found that there are many minor technical implications which I initially struggled to grasp, such as how pushing stack parameters defaults to 4 bytes at a time.

```
for (int i = 0, i < original.Length(), i++) {

  if (original[i] < 97 || original[i] > 122)    // Only encrypt lowercase
    continue;

  char replacement = key[original - 97];        // Find replacement

  original[i] = replacement;                    // Replace
}
```

Fig. 2 - Encryption pseudocode

```
for (int i = 0, i < original.Length(), i++) {

  if (original[i] < 97 || original[i] > 122)    // Only decrypt lowercase
    continue;

  for (int j = 0, j < 26, j++) {                // Foreach char in key
    if (original[i] == key[j]) {                // if it matches orig
      original[i] = j + 97;                     // replace with original
      break;
    }
  }
}
```

Fig. 3 - Decryption pseudocode

Conclusion

Looking back, I wouldn't change much about my implementation. I still think that I could have a more solid understanding of the stack though, and it would have been especially helpful to spend more time in preparation by studying stack offsets.

My advice to future students in this class would be to learn to debug memory effectively. I found that observing the system stack in bytes synthesized my research in this class by providing a tangible representation of MASM processes. Being able to debug memory and registers washed away most of my confusion as I stepped through each line of code and watched the direct results in memory.