David Alummoottil
PA3 – Mystery readme

Given a program written in x86 AT&T 32-bit format assembly language, I needed to figure out what the program did. Although I had a good understanding of assembly language, I decided to use a debugger, gdb, to see what happened to the input as it moved along in the program. After figuring out the general idea of the parts in the program, I also manually wrote by paper how some things were moving from register to register in order to understand where values were being stored and how they were being manipulated. Although this took some time, I was able to understand how assembly language worked to a greater extent. After some time playing around with the debugger and manually working with the code, I was able to piece together that the program was calculating what the nth number in the Fibonacci sequence is. One part that confused me was the loop I encountered in the main while using gdb. It kept running the loop and I had to figure out how to skip the loop to continue and figure out what the rest of the program was doing. After writing a C program that I believed followed the assembly code. After compiling and creating two versions of the assembly for the C program, one optimized and one not, I could see some differences that help the program run faster. I could clearly see that most changes resulted in the optimized function having less lines of code, which means it compressed actions and did things more efficiently than the unoptimized code. In the add function there was only one modification. Instead of putting both parameters into registers and adding with leal, the compiler made it move one parameter into a register, then used add to add them together into the eax register to be returned. In the dosomething function, initially you see that less memory is allotted for variables so you can predict the compiler is predicting to you use less space than when it is unoptimized. In the first compare instruction, less time is taken in the optimized because the value in the array is compared directly to -1 rather than moving that value into a register then comparing it. In general, this is a common trend seen. Instead of moving values into registers and comparing the registers, the compiler compares the values more directly and with fewer operations. It often uses leal instead of add or sub because leal can take two arguments and move it into another designated destination which would take extra instructions to do in unoptimized code. A difficulty I personally had at times was that I would get hung up on miscellaneous instructions the compiler does during the compilation process. Instead of focusing on the big picture, I would think too much about each instruction and put too much importance into instructions that were not important.