AOI Exercise 1 Report

Guy Danielli – 315657338

Sample Execution

The result of the program for user "315657338" is "spider".

Analysis

With $a-z, A-Z, 0-9$ possibly in the password, there are 62 possible characters. Theoretically, a brute-force attack would require a maximum of $O(Z^n) = 62^{32}$ attempts (n is the length of the password and Z is the number of possible characters at each point of the password). In this case specifically (with the right password being "ghost"), it would have required 62^6 requests, which is equivalent to about 2^{35}. In my case, the attack requires $O\left(C * n * \sum_{i=1}^{\log_2 Z-1} 2^i + Z + N\right) = O\left(C * n * 2^{\log_2 Z-1+1} + Z + N\right) = O((C * n + 1) * Z + N)$, which in this case corresponds to $(2 * 6 + 1) * 62 + 32 = 838 < 2^{10}$ requests. Here C is a configurable coefficient that indicates how many requests should be sent per measure (the measured time is averaged over C requests). N is the maximum password length. During experiments I usually only needed C to be 1, but for robustness I used a greater value. Note that a naïve implementation would require $O(C * n * Z + N)$, but my method requires a much smaller C than the naïve one.

Optimizations

I've made several optimizations for the program to run as fast as possible. The first and simpler optimization is that, when looking for the last character of the program, instead of finding it based on the RTT of several requests, I base my knowledge on the response of the server. In other words, the program simply queries the server for each possible character once until it finds the correct password. That way I only need to send 1 request per guess for the last character instead of C requests.

The second optimization is in the way the program finds each character of the password. Instead of sending C requests for each character and then finding the character which had the maximum average RTT, I used a method similar to binary search: each character index in the password there are several iterations. At each iteration, the program measures the RTT for each possible character, then removes the half of the characters that had the smallest time values from the group of possible characters. This operation is repeated until the group of possible characters remains with a single character, which is then returned as the correct one.

This optimization lets us repeat the measurements a smaller number of times for each character index in the password, while still having high confidence in our guesses (it is likely that the correct character won't have the maximum RTT in a single request because of noise, but is unlikely that it will have smaller RTT than half of the characters overall). Moreover, to not waste previous results, the measurements accumulate for each of the surviving characters between iterations, which gives more higher confidence in the result.

To ensure that the program does not fail during testing in the submission server, instead of taking out half of the characters at each iteration, I take out a quarter of the characters. That

way the chance of failing is smaller, and it's a minor slowdown because that way I could define a smaller C (1).