Jimmy O'Rourke
jorourke
20419748

ECE458 Assignment 2

1. Proof of Work

The source code for this is in server.cpp and client.cpp and the server and client executables can be built by running make server and make client, respectively (or make all). The server reads 16 bytes (128 bits) for R from /dev/urandom, and then reads 2 bytes for P. These are hex-encoded and transmitted to the client for the hashing to be performed. The client decodes R and P and keeps reading 128 bits from /dev/urandom and computing its hash until it finds a string with a hash beginning with P. The client gets the time before and after completing the hashing in order to measure how long it took. If the time is less than the minimum of 100ms, it sleeps until it has been that long, at which point the result is transmitted back to the server where it is verified as correct and not arriving too soon (in less than 100ms). If it verifies correctly and the time was correct, the server transmits "Ok" back to the client. If the client takes longer than 1s to respond, the server simply closes the connection. The client also checks that it has not taken too long to generate the hash, and if so it gives up. Reading from /dev/urandom is used as it is more validly random than calling rand() since it based on relatively unpredictable events happening in the machine, such as the frequency of interrupts or disc accesses and as such produces a more valid random string as a result.

The lengths of P and the timeouts were chosen empirically as follows.P was chosen to be 2 bytes in length as this was large enough that the client must do enough hash iterations that the time it takes is easily distinguishable as being larger than simply responding immediately, taking into account the possible variance in socket latency. At a length of 2 bytes for P, the time it takes to generate a correct hash and respond to the server typically ranges from 0.013s to 0.7s., while responding immediately took on average between 0.0004s and 0.0008s. Increasing P to 3 bytes led to average times in the range of 20s which is not realistic, in that it should not take a client 20 seconds to establish a connection with the server. The 0.7s high end response time of the 2 byte length is reasonable, in that a delay of less than a second in establishing a connection is not noticeable to the end user on the client side. This noticeable latency criterion as well as the range of observed values were what were used in deciding upon the server's timeout limit of 1s, after which the server will close the connection. Connection latency of greater than 1 second becomes noticeable to the user. The minimum response time was chosen as 100ms, as any immediate responses are faster than this and will thus be rejected, and if the client does compute the hash correctly, it typically takes at least this long, so the client does not usually have to waste any time waiting for this minimum time to expire before responding.

2. Timing Attack on Passwords

The source code for this is in time_attack.cpp and the time_attack executable can be built by running make time_attack. The program cracks the password interactively, and takes a partial password as command line argument. Passing no argument will try an empty string and thus return the first letter of the password, which can then be used for the next iteration. In order to determine the password, the

program calls rtdsc() before and after calling the check_password() function, in order to measure the duration of the string compare test for each letter. The program iterates through the alphabet, testing each letter enough times until it obtains a 95% non-overlapping confidence interval for the letter with the highest mean duration. The letters of the alphabet are randomly shuffled between each iteration to reduce the hardware effects such as cache hit and miss vectors from skewing the results.

The program uses a vector data structure, such that for each letter it holds the sum of the durations from each iteration, and the square of the sums. These values are used to calculate the mean, variance and average standard deviation of the averages for each letter, using the formulas described in the assignment guidelines. Keeping track of these values makes it possible to use confidence intervals to determine when enough iterations have been performed to predict the next letter of the password. The program must determine when the 95% confidence interval of the letter with the highest mean duration does that overlap with any others. To do so, the program keeps track of which letter has the highest mean duration and what that maximum mean is. It also keeps track of the maximum endpoint of the next highest confidence interval, meaning that out of the remaining letters (all but the one with the max mean duration) it keeps track $of\ max\{mean[i] + 1.96 * std\_dev[i]\}$. The confidence intervals thus do not overlap when letter with the max mean duration has a confidence interval with a minimum endpoint that is greater than this next highest interval, ie $(max\_val.mean - 1.96 * max\_val.std\_dev) > max\{mean[i] + 1.96 * std\_dev[i]\}$. The reason we can't just keep track of the values for the top 2 maximum means is that another value could have a very large standard deviation such that its confidence interval extends beyond that of the 2nd highest letter.

Once enough iterations have been performed that the confidence intervals do not overlap, the letter with the highest mean is output to the console as being the prediction for the next letter of the password.