Nolan Pofi

SHA-1, HMAC, and Blum-Goldwasser

For the final project I was entrusted with writing the SHA-1 hash algorithm, the HMAC algorithm for using a hash as a MAC for message authentication, and converting the Blum-Goldwasser algorithm, from the third homework, to our format for use as an encryption/decryption algorithm. I worked on and implemented them in that order as a functioning SHA-1 was necessary for HMAC to work and I figured Blum-Goldwasser would be a quick rearranging so I saved it for last however, it would prove to be more than just a quick fix. While our project was programmed in python I was the only member of our group to code the homework in python so I also submitted all of my code for the group to use as a reference when making their parts.

I implemented SHA-1 using information from the lecture notes and from the textbook. The function I wrote takes a message of any length and shortens it into a 160-bit hash value. It begins by specifying the initial values of the five 32-bit blocks A,B,C,D,E which match the values outlined in the lectures. It then pads the message so that its length in bits is a multiple of 512 and such that the last 64-bits give the overall length of the non-padded message. The fun part then begins by breaking the message into 512-bit chunks and then operating on each one chunk by chunk. This begins by breaking the 512-bit chunk into sixteen 32-bit chunks and then expending into eighty 32-bit chunks all based off of the bits stored in previous chunks save the original sixteen. Then each of those chunks change the values of the A,B,C,D,E chunks and after all eighty chunks are used A,B,C,D,E are then added to the original values saved in A,B,C,D,E all of these addition are mod 2^32 however so as to keep them 32-bits in length. Then the rest of the 512-bit chunks are worked on the final values in the A,B,C,D,E blocks are combined into one

160-bit long value. The expansion to eighty 32-bit chunks and the subsequent operations all add together for a nice avalanche effect as one small change is constantly propagated through the whole operations allowing for hashes to be quite different despite being very similar at the beginning. The string “Alice and Bob” has a hash value of 0x8439cad63491761d42fbb8e31b03a1f1816d0f66 while the string “Alice and Rob” has a hash value of 0xe048bb78fff5c5780c2bf11aac72564e4543c575 despite having only one-character difference the hash values don’t look similar at all.

I then implemented HMAC using the SHA-1 function I just previously described. This was the easiest and least time consuming of my parts on the project as it was quite simple. In fact, the bulk of the time I spent on it was less on the algorithm and more on the class that uses it and is used in the actual handshake. The HMAC function takes two values as input a key and the message to be hashed. It then checks to see if the key handed to it was larger than 512-bits, the block size used by the SHA-1 hash function, and if so it hashes it down using SHA-1. Then the key, which is now less than 512-bits, is padded to be 512-bits long. The Opad and Ipad are then constructed as they are some constant appended to be 512-bits long and then Xor’d with the key. The Ipad is then concatenated with the message and the resulting value is then hashed. The new hashed value is then concatenated to the Opad and that value is hashed to give the result. The HMAC class is just a simple class used in our interface that calls upon the HMAC function that I wrote. The class only took me longer due some communication delay so I had trouble formatting it exactly how Daniel needed it as he was working on the handshake.

Last but not least I worked on formatting the Blum-Goldwasser algorithm used in homework 3 so that it works in our handshake. I didn't factor in that in the homework many of the initializers were given and that in our implementation I would have to generate those values. I began by generating p and q which have to be large prime numbers such that p=q=3 mod 4. I quickly came up with generating a random number and multiplying by 4 and then adding 3 to it thus ensuring the random number is 3 mod 4. However, I then had to prove that this new value was prime thus I lucked out by having implementing the Miller-Rabin primality test for the exam one makeup so I brought it over and set up a loop that generated a number and checked if its prime and if it was then it broke out of the loop. I did the same for the q value however I also added a check to make sure it never equaled p. I was worried that this method would take a while to generate two values but it was surprisingly quick. Next came generating the a and b values used for decryption however I solved this by using p^(q-2)mod q to get the inverse of p mod q and q^(p-2)mod p to get the inverse of q mod p. Generating the x_0 used in the encryption was trivial as well as I did r^2 mod n where n = p*q and r is a random number 1< r < n. The actual encryption and decryption algorithms were trivial as I simply had to copy and paste from the homework and rename a few variables. Though there was much time spent getting the algorithm to match the format we had the other algorithms matching as it was fairly different but I eventually got it all squared away so that two BlumGoldwasser objects could exchange messages after they share their public keys which is just n.