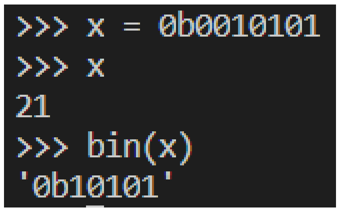
The implementation for this project tried to implement as many cryptosystems as possible, which was inherently a very ambitious goal. The struggle came when negotiating between the key exchange and then the actual cipher suite. When our group worked towards establishing a key exchange we wanted to have as many options available for not only white hat credit but to have the option for a server to pick the level of security wanted. With so many cryptosystems selected it became overwhelming to keep track of. Plus once a key exchange was selected and the session key was established the process of encrypting and decrypting the messages between both client and server.

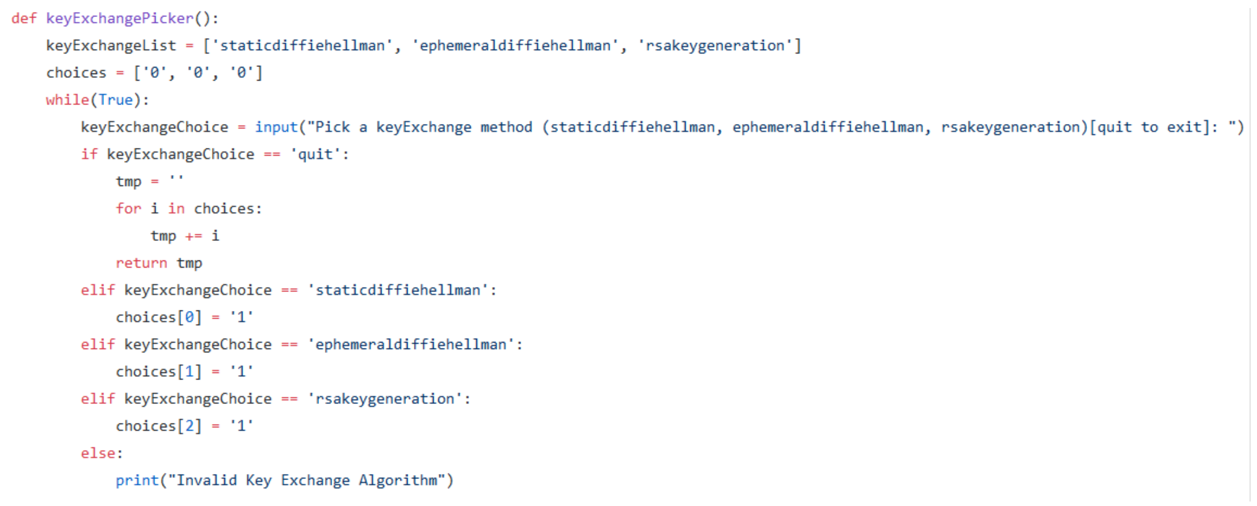
Another struggle was with the SHA implementation. Python has a lot of overhead which proved to be unexpectedly troublesome, because Python would want to remove any leading 0's in a binary string to make numbers look pleasing, which is great in some cases. However, in a case where fixed bit lengths are a must, the leading bits are important to keep.



```
>>> x = 0b0010101
>>> x
21
>>> bin(x)
'0b10101'
```

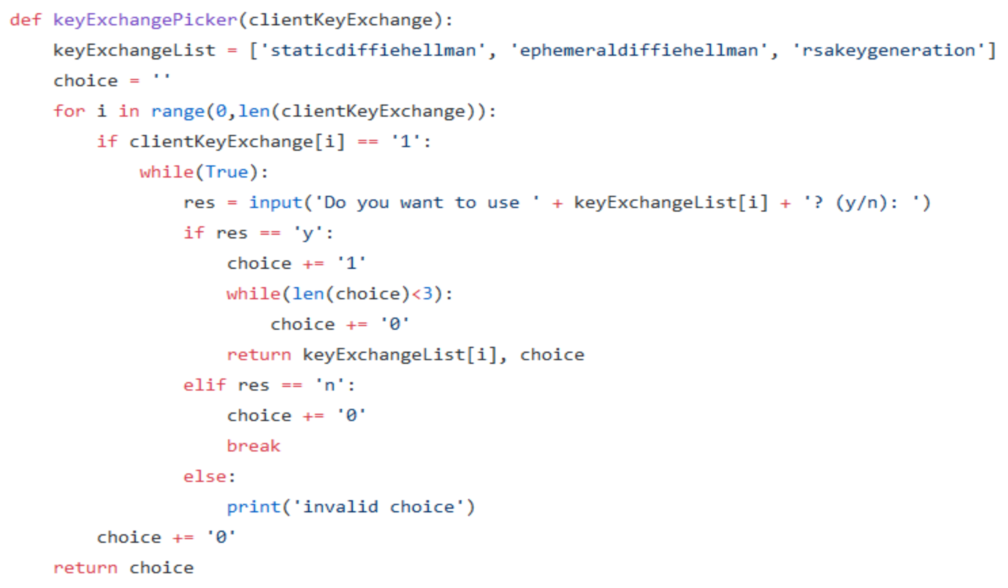
In the image above you can see a previously 7 bit long string become 5 bits long, which would misinterpreted by many cryptosystems required for SSL/TLS to perform as needed. A solution that was found was a bit string and bit array package that python has, similar to that of java's byte and byte[] data type. All of this led to a lot of bugs in functions like SHA1 and DES. Given more time this bug could have been patched and not have a 1 bit error within SHA1.

Another major note was that since so many systems were implemented, the security of the entire system suffered. I don't want to go into the details of what is at risk for the inevitable black hat team reading this paper, but the goal was for a server to have an option of what was wanted to be used, or completely back out of a session all together if they wanted to.



```
def keyExchangePicker():
    keyExchangeList = ['staticdiffiehellman', 'ephemeraldiffiehellman', 'rsakeygeneration']
    choices = ['0', '0', '0']
    while(True):
        keyExchangeChoice = input("Pick a keyExchange method (staticdiffiehellman, ephemeraldiffiehellman, rsakeygeneration)[quit to exit]: ")
        if keyExchangeChoice == 'quit':
            tmp = ''
            for i in choices:
                tmp += i
            return tmp
        elif keyExchangeChoice == 'staticdiffiehellman':
            choices[0] = '1'
        elif keyExchangeChoice == 'ephemeraldiffiehellman':
            choices[1] = '1'
        elif keyExchangeChoice == 'rsakeygeneration':
            choices[2] = '1'
        else:
            print("Invalid Key Exchange Algorithm")
```

This is a typical method picker that a client would see and be able to pick all the key exchange systems they want to implement. After all the choices are made than a binary string was generated and sent to the server similar to that of a header in a packet with parameters of communication.



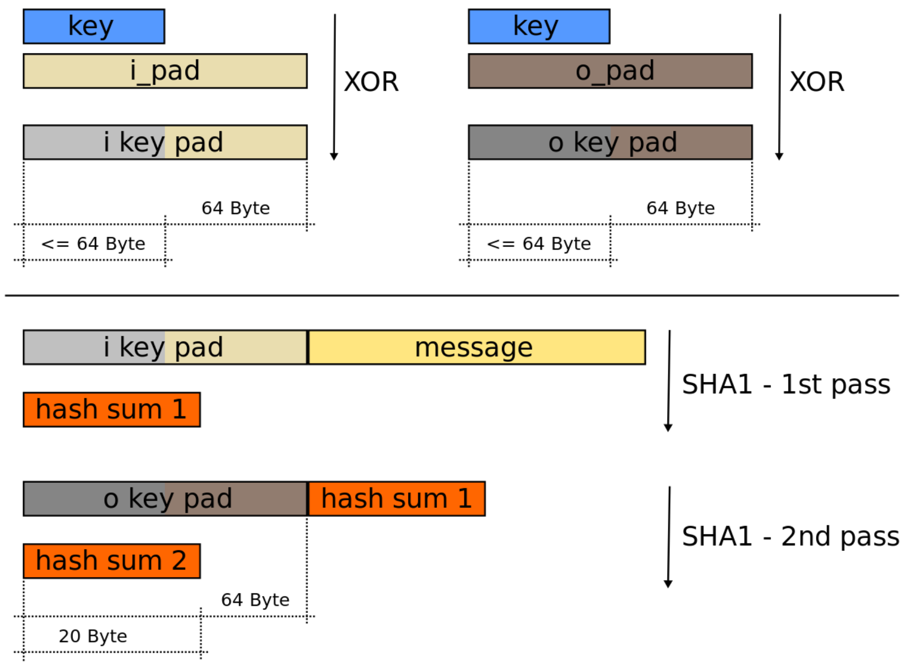
```
def keyExchangePicker(clientKeyExchange):
    keyExchangeList = ['staticdiffiehellman', 'ephemeraldiffiehellman', 'rsakeygeneration']
    choice = ''
    for i in range(0,len(clientKeyExchange)):
        if clientKeyExchange[i] == '1':
            while(True):
                res = input('Do you want to use ' + keyExchangeList[i] + '? (y/n): ')
                if res == 'y':
                    choice += '1'
                    while(len(choice)<3):
                        choice += '0'
                    return keyExchangeList[i], choice
                elif res == 'n':
                    choice += '0'
                    break
                else:
                    print('invalid choice')
        choice += '0'
    return choice
```

Here we have the server processing the choices made by a client, and picking which method they wish to use. The goal of this was to present the server with the options in cascading order of security, but time pressures led me to just presenting them as they were implemented. Similar methods were created for picking cipher suites and the hash algorithm, with the same goal of keeping security in mind.

Another aspect of the project that played a major role is actually picking valid numbers for key encryptions, hashing, and key exchanges. Within the classroom the numbers picked for key exchanges and other cryptographic systems were already chosen at the start. But in this case we had to, not only find valid combinations of primes and relative primes, but large enough to prevent brute force attacks. These numbers also needed have high entropy, and independent of each other. The more independent these choices are result in less information an attacker could generate. Python has a module called secrets that promises high level entropy and for dynamic bit lengths. This is ideal because a lot of different systems have fixed values that make it difficult to accommodate for all of them.

The actual sending of messages between two people was fairly trivial. Python offers a lot when it comes to opening and communicating over a socket. Our implementation actually provides for communication over a network that isn't just someone's localhost (127.0.0.1), as long as a port is communicated between the two parties in the beginning. Something that was troublesome was that the socket isn't unidirectional but the socket cannot send messages while it is receiving. This makes building a chatroom, which was what the team wanted, difficult. The client would have to take a message, encrypt it with its negotiated cipher suite and then send it to the server who would decrypt the message and then be able to send its response. This doesn't allow for a person sending multiple messages at once. Maybe with threading "instant" messaging could be implemented but for the sake of the project completing, only one person can communicate and the other listens before switching.

The MAC was also an implementation that we wanted to implement, especially the hash message authentication code (HMAC). We were able to setup the HMAC which could be appended to messages to show that a person was who they say they were. With our SHA1 implementation we could implement this with block sizes of 64, however the bug with our SHA1 prevented a full implementation of this feature.



This was the HMAC schema that we followed which provided security on the premise that finding a collision would be extremely rare with a size so big generated by SHA1. There is also a security issue with this that is very well known. SHA1 has become an unsecure hashing algorithm by Google as of 2017. Where they were able to find a collision between two pdfs reliably. This was said to be 100,000 times faster than a brute force attack or the birthday paradox attack we learned in class. However, this still requires a lot of processing power and in our class I doubt a Lenovo will able to get a lot of information in the week provided for the black hat phase of the project.

This projects was a lot of fun to work on, and I wish there was less of a time restraint. If there was more time then teams could add their own flare to the project and develop some cool applications that go beyond a rudimentary chatroom or echo system for a server. The tight time constraints caused our group to rush through a lot of the implementation which will cause us to suffer in the black hat phase. I do believe though there was a lot of hard work that might go unnoticed without reading this report, and a lot that was learned by implementing all of these cryptographic systems on our own. When it comes down to it the internet is definitely as secure as you make it, and can be really interesting to see the lengths people go to try and make it as such.