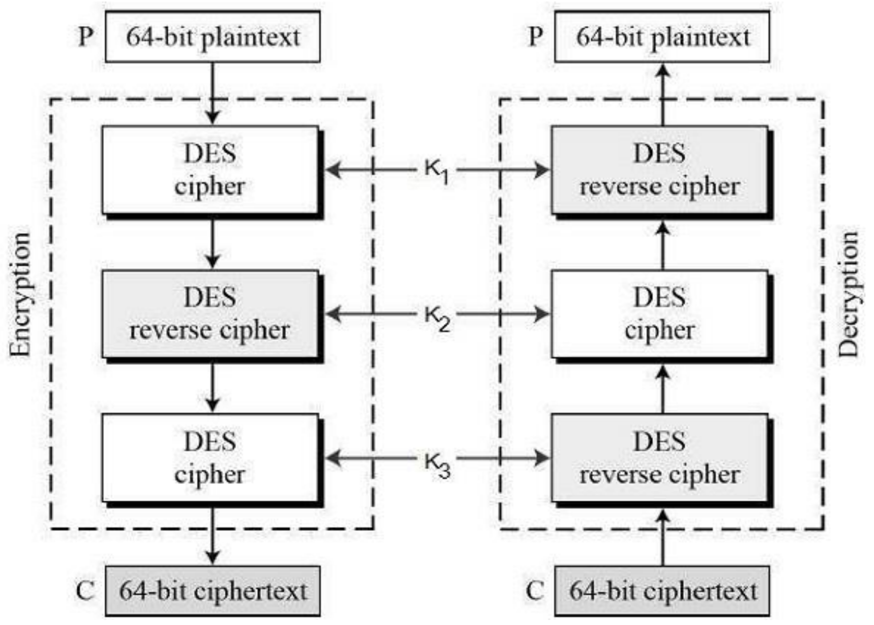
Ryan Berlin
Cryptography and Network Security I
Final Project White-Hat Writeup

For our group's final project, we aimed to implement our own version of an SSL protocol that allows two users to send messages over a secure channel. The protocol allows the two parties to negotiate which crypto algorithms to use during the communication, which have all been implemented by from scratch. Our crypto suite currently supports RSA, Cramer Shoup, Triple DES, and SHA-1 for hashing. Charlie handled the protocol, Nick implemented RSA and Cramer Shoup, and my contribution to the project was implementing Triple DES and SHA-1.

Triple DES Implementation

Originally, we had only planned on using the Toy-DES implementation we did for homework 1. Since this would have been a gift for the black hat team, I took it upon myself to expand this into Triple DES. This uses a 168-bit symmetric key, and while it is not used as much in the real world as compared to AES, it is still a huge upgrade in security when compared to standard DES's 56-bit key, and especially to Toy-DES's 10-bit key. Triple DES is essentially just 3 rounds of the standard DES algorithm, using three different keys. This can be seen in the diagram below.

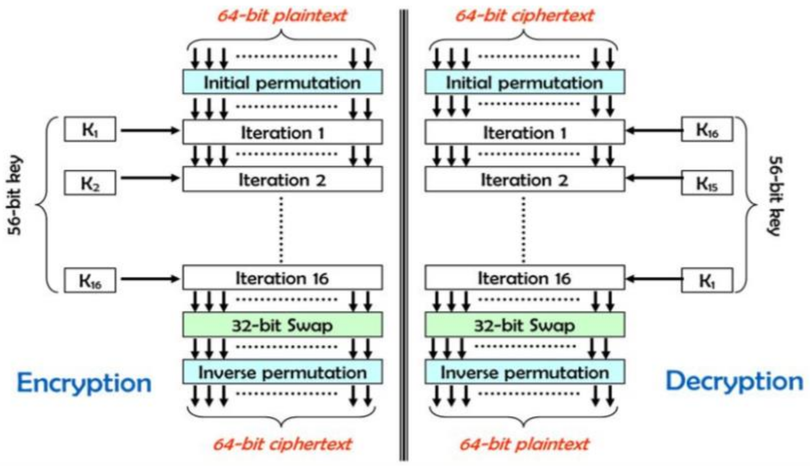


Ryan Berlin
Cryptography and Network Security I
Final Project White-Hat Writeup

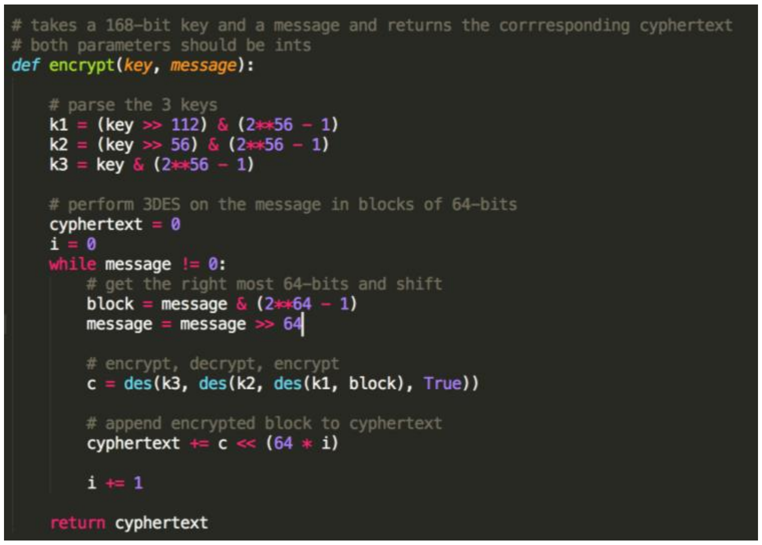
Like DES, Triple DES encrypts and decrypts messages in 64-bit blocks. The block is then encrypted, decrypted, and encrypted again using DES with different keys. The different keys for each step come from the 168-bit key for Triple DES, which is broken into 56-bit keys.

To implement 3DES, I would first need to implement DES by upgrading my Toy-DES program, which presented my first design challenge. For Toy-DES, I did most of my calculations on strings of bits, which I didn't love doing the first time, so when I went to implement DES I ended up re-writing a lot of code using bitwise operators on numbers. I found doing it this way to be a lot cleaner, and it saved me from converting back and forth from strings of bits to numbers.

For my implementation of DES I first hard-coded all of the permutations and s-boxes needed for the calculations, which I found online. Since the encryption and decryption method for DES is very similar, I chose to write a single function that could perform both operations.



The function performs an encryption by default, but takes an optional parameter that can be set to True to allow for decryption. In this case, all that needs to be done is reverse the order of the subkeys used in each round, as seen in this diagram. In addition to this function, I wrote a few more helper functions to compute the subkeys, compute the round function, permute bits according to a permutation table, and substitute bits according to an s-box. After DES was implemented, making Triple DES was easy. I simply parsed the key into the three DES keys, broke the message into blocks of 64-bits, and then performed the three rounds of encryption on each block. My Triple DES encrypt function can be seen below.



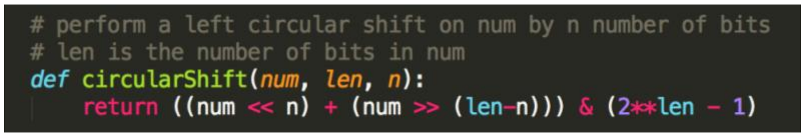
```
# takes a 168-bit key and a message and returns the corrresponding cyphertext
# both parameters should be ints
def encrypt(key, message):

    # parse the 3 keys
    k1 = (key >> 112) & (2**56 - 1)
    k2 = (key >> 56) & (2**56 - 1)
    k3 = key & (2**56 - 1)

    # perform 3DES on the message in blocks of 64-bits
    cyphertext = 0
    i = 0
    while message != 0:
        # get the right most 64-bits and shift
        block = message & (2**64 - 1)
        message = message >> 64

        # encrypt, decrypt, encrypt
        c = des(k3, des(k2, des(k1, block), True))

        # append encrypted block to cyphertext
        cyphertext += c << (64 * i)

        i += 1

    return cyphertext
```

SHA-1 Implementation

My next task was to implement the SHA-1 hash algorithm, which takes in a message of any size, and converts it to a 160-bit hash. For this algorithm, I followed the guidelines laid out in RFC 3174, from the padding technique to using the suggested constants h0-h4 as initial hash values:

```
H0 = 67452301

H1 = EFCDAB89

H2 = 98BADCFE

H3 = 10325476

H4 = C3D2E1F0.
```

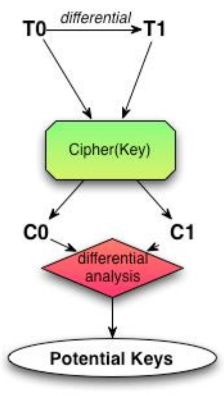
SHA-1 hashes the input in chunks of 512 bits, so the first step is to add padding to the input to make sure that the length is divisible by 512. This is done by appending a 1 to the end of the input, followed by a certain number of 0s, and then 64 bits representing the original length of the input. The arbitrary number of 0s is used to make the length divisible by 512. Since we append the length of the original input in 64 bits, the max length of the input can be 2^{64} bits. Each chunk of input is then sent through the main computation loop, and combined to our totals h0-h4, which are concatenated into our final message digest after the whole input has been hashed. For this implementation, I used one helper function that allowed me to perform a left circular shift on any size input by any number of bits. This function can be seen below.



```
# perform a left circular shift on num by n number of bits
# len is the number of bits in num
def circularShift(num, len, n):
    return ((num << n) + (num >> (len-n))) & (2**len - 1)
```

Security Considerations

As I briefly touched upon earlier, Triple DES, while more secure than its smaller key size implementations, is still not widely used for symmetric key encryption anymore. AES has been recognized as a much more secure standard than DES. In the written part for homework 1, we



saw that DES is susceptible to differential cryptanalysis, a type of chosen plaintext attack. A differential cryptanalysis attack would work by first constructing a Differential Distribution table for S_0. Then, assuming we know two inputs S_E and S_K such that $S_E \oplus S_K = S_I$, and the output XOR is S_O. We can then look to the table for row S_I and column S_O, and this will give us a list of different input combinations. Finally, we XOR each of these values from the table with S_E and S_K to get a list of possible keys, removing duplicates from these lists. This process can be repeated with different values for S_E and S_K, and the true key will be in both of the resulting steps, so by repeating this process multiple times, we can narrow down the set of possible keys until there is only one true key remaining.

The security of SHA-1 largely depends on the size of the hash, 160 bits. The birthday problem suggests that SHA-1 should then have 80-bit security against a brute force attack, however it has since been proven to be even weaker than this. As early as 2005, attacks on SHA-1 have consistently taken less than 2^{69} operations. Now, attacks have become so strong against this algorithm that it can be broken in about an hour on an average PC. That all being said, it makes sense why this algorithm is no longer in use.