Lab 06 - Stream Ciphers in Software

Part 01)

RC4 was one of the first stream ciphers to be implemented in software. Although it didn't use a feedback shift register it still was very fast and simple. Designed by Ron Rivest in 1987 it was formally named Rivest Cipher 4, but became widely known as Ron's Code 4. We will hear more about Rivest in lecture, but suffice it to say now he is one of the few most influential characters in modern cryptography.

For years it was the default cipher for SSL/TLS connections and was used in the Wired Equivalent Protection WEP security for WiFi. Unfortunately, RC4 does not create a truly random keystream and was removed from the acceptable TLS cipher suites.

To understand the shortcomings of RC4 we must understand how it works. RC4 consists of 2 main parts: a key scheduling algorithm (KSA) and a pseudo-random generator (PRG). As it turns out, the first bytes generated from the random generator are strongly non-random. This is known as having a statistical bias. In addition, the KSA also has known weaknesses.

Part 01 of the lab needs to be done on the Ubuntu machine you have inside of ISELab. It has the correct version of OpenSSL (1.1.1) which still includes RC4.

1. Let's start by using OpenSSL to encrypt a message using the old RC4 stream cipher. As you remember from lecture, RC4 doesn't have a nonce or initialization vector. It only needs a key. For this example, use the 13-byte hex key of 6a61636f623d616d617a696e67. Create a file that contains the message "**Strike me down and I will become more powerful than you could possibly imagine**" Then issue the command below. **<u>You will upload the encrypted message to Canvas.</u>**

   Don't worry about the padding message. I don't want you to go through the pain of typing a 128 byte key so rc4 is padding it with zeros for you.

   ```
   cpre331@desktop:~/Documents/rc4$ openssl enc -in plaintext -out ciphertext -K '6a61636f623d61617a69e67' -rc4
   hex string is too short, padding with zero bytes to length
   ```

2. To decrypt you just add the -d flag.

   ```
   cpre331@desktop:~/Documents/rc4$ openssl enc -d -in ciphertext -out original_message -K '6a61636f623d61617a69e67' -rc4
   hex string is too short, padding with zero bytes to length
   ```

3. You can also use echo and pipe to send a message of your choice into openssl and xxd will allow you to convert hex to ascii. It can't be Happy midterms! Please keep the message reasonable. You will be taking a screenshot of it and turning it in as part of your lab report.

```
cpre331@desktop:~/Documents/rc4$ echo -n 'Happy midterms!' | openssl enc -K '6a61636f623d61617a69e67' -rc4 | xxd
hex string is too short, padding with zero bytes to length
00000000: ea4c b4dc fc0f 91ac b33e 1cdc 5bf4 d6    .L.......>..[..
```

4. To decrypt you can send that back into the decrypt function. **Include a screenshot of this step in your lab report.**

```
cpre331@desktop:~/Documents/rc4$ echo -n 'Happy midterms!' | openssl enc -K '6a61636f623d61617a69e67' -rc4 | openssl enc -d -K '6a61636f623d61617a69e67' -rc4 |
xd
ex string is too short, padding with zero bytes to length
ex string is too short, padding with zero bytes to length
0000000: 4861 7070 7920 6d69 6474 6572 6d73 21    Happy midterms!
```

5. As you know, using the same key to encrypt multiple messages in a one-time pad is a bad idea. Likewise, it is a bad idea in RC4. However, some systems setup an initialization value to prepend to the key to have 3-bytes that changed. That way every new plaintext was encrypted with a new IV. This also was a bad idea.

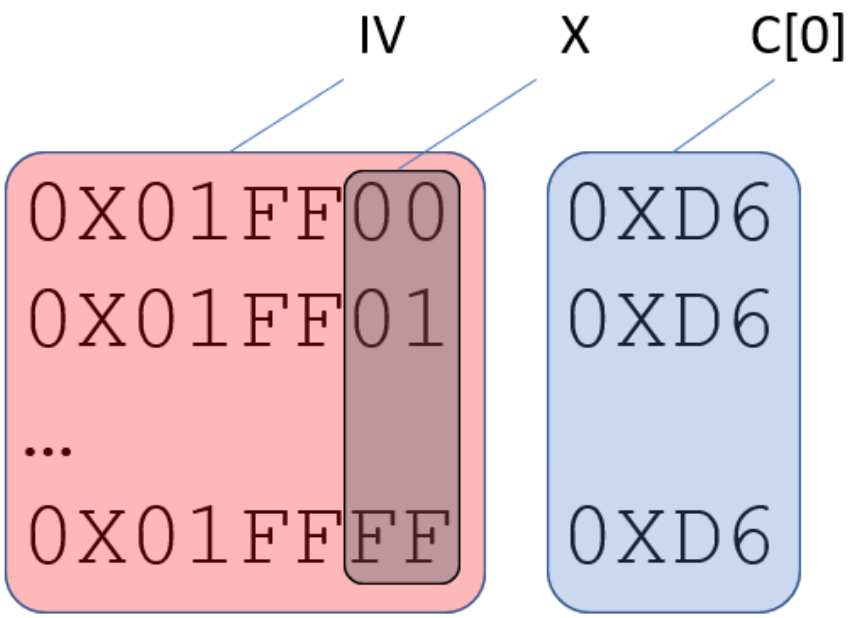
```
cpre331@desktop:~/Documents/rc4$ openssl enc -in plaintext_1 -out ciphertext_1 -K '01FF006a61636f623d61617a69e67' -rc4
hex string is too short, padding with zero bytes to length
```

6. What you are going to do now is do a key-recovery attack. You are going to assume:
    a. The RC4 key has a 3-byte initialization value (iv) prepended to a 13-byte long-term key similar to what you see above.
    b. The initialization vector increments with every new encrypted message.
    c. The communication system from which these packets were captured is well-structured with a constant header of at least one byte which you will consider as message 0 (m[0]).
    d. You will be provided many encryptions of that constant header (m[0]). This is as if the attacker was able to eavesdrop on the communication system and captured traffic that holds multiple copies of the same ivs.
7. Connect to repo.331.com and scp the files in the lab06 directory. repo/repo are the credentials you need. For this part of the lab use the files stored in lab06/part_01. A python template has been provided for you to make things easier.

```
cpre331@desktop:~/Documents/lab06_test_download$ scp -r repo@repo.331.com:/home/repo/lab06/* .
repo@repo.331.com's password:
bytes_07FFXX.txt
bytes_0AFFXX.txt
bytes_0FFFXX.txt
bytes_01FFXX.txt
bytes_04FFXX.txt
bytes_02FFXX.txt
bytes_0DFFXX.txt
bytes_05FFXX.txt
template.py
bytes_06FFXX.txt
bytes_03FFXX.txt
bytes_0CFFXX.txt
bytes_0EFFXX.txt
bytes_0BFFXX.txt
bytes_08FFXX.txt
bytes_09FFXX.txt
chacha20_quarter_round_skel.py
cha_ciphertext
```

8. These files contain the value of initialization vector and the corresponding output of one-byte of ciphertext when each initialization value (3-bytes) is prepended to the long-term key (13-bytes), then used by RC4 to encrypt m[0].

   The text files below can be visualized as seen below. The three-byte IV is on the left and the ciphertext C[0] is on the right.



9. There are three flaws that will help you recover the long-term key and be able to decrypt the initial message.
   a. Flaw #1 - For any byte x, using the iv = 01FFx results in a keystream where it is highly probable the first byte equals x+2. In other words, you will perform this operation on each line where the IV resembles 01FFX (X ranges from 00 to FF). The most common value out of these 256 computations will be your guess for m[0].

      ```
      C[0] XOR (X+2 % 256)
      ```

   b. Flaw #2 - When iv = 03FFx there is a high probability that the first keystream byte is x + 6 +k[0]. K[0] is the first byte of the long-term key (13-byte key). In other words, you will perform this operation on each line where the IV resembles 03FFX (X ranges from 00 to FF). The most common value out of these 256 computations will be your guess for k[0].

      ```
      (C[0] XOR m[0]) - x - 6
      ```

c. Flaw #3 - For i ranging from 0 to 12, the iv = zFFx, and z is the hexadecimal value of i +3, the first byte of keystream is equal to x + d[i] + k[0] + …. k[i] where d[i] = 1 +2 +....+(i+3). For example, if iv = 04FFxx the first keystream byte is equal to x + 10 + k[0] +k[1] where k[1] is the second byte of the long-term key.

   To derive k[1] we would use:
   `K[1] = (C[0] XOR m[0]) - x - (1+2+3+4) - k[0]`

   To derive k[2] we would use:
   `K[2] = (C[0] XOR m[0]) - x - (1+2+3+4+5) - k[0] - k[1]`

   To derive k[3] we would use:
   `K[3] = (C[0] XOR m[0]) - x - (1+2+3+4+5+6) - k[0] -k[1] -[2]`

   So on and so forth. Keep in mind that d[i] may need to be in base 10. Subtracting 15_{16} is much different than 15_{10}. In addition, we are using unsigned 8-bit arithmetic. There should be no negative values and no values greater than 255. Using modulus on the operations can help.

10. Now to the actual recovery. Each of the above flaws are included as functions to be written in the skeleton code.
11. Use Flaw #1 compute 256 values of c[0] XOR (x+2). Choose the most frequent one as your guess for m[0].
12. Use Flaw #2 to recover k[0]. Use your guess from m[0] from the step above to compute (c[0] XOR m[0]) -x -6. Again, find the most frequent value and that is most likely your k[0]
13. Still using Flaw #2 find the most frequent value of (c[0] XOR m[0]) -x -10 - k[0].
14. Finally, use Flaw # 3 to find all other bytes of the key.
15. Example of your final output would look like the text below. You probably will have counts and other values you may or may not want to include.

    Guess for m[0]: 0x6
    Guess for k[0]: 0x61
    Guess for k[1]: 0x6c
    Guess for k[2]: 0x6d
    Guess for k[3]: 0x6f
    Guess for k[4]: 0x6e
    Guess for k[5]: 0x64
    Guess for k[6]: 0x62
    Guess for k[7]: 0x75
    Guess for k[8]: 0x74
    Guess for k[9]: 0x74
    Guess for k[10]: 0x65
    Guess for k[11]: 0x72
    Guess for k[12]: 0x73

16. Since we used printable ascii values and English word(s) as the key you may want to convert your hex values to ascii to see if your answer is logical.
17. **<u>Turn in your code documenting your code for each of the three flaws. Also, turn in the first byte of message and the key in your lab report.</u>**

Part 02)

While ChaCha20 is a well-known modern stream cipher created by Daniel J. Bernstein, it isn't often used for encryption by itself. It is usually used as one of two components in the ChaCha20-Poly1305 encryption algorithm. However, it can be used for encryption as a stream cipher as you will see below. It is also commonly used as a pseudo-random number generator.

As shown in the lecture, the generator of the keystream bits (or the random number bits) come from quarter-round operations. Each quarter-round is a combination of ARX (Addition, Rotation, XOR). You will be provided code that implements ChaCha20. You will be using this code to look at the diffusion of bits in just the first odd quarter round.

- Operation is quarter-round

  QR(a,b,c,d)

  a ⊞= b; d ⊕= a; d <<< 16

  c ⊞= d; b ⊕= c; b <<< 12

  a ⊞= b; d ⊕= a; d <<< 8

  c ⊞= d; b ⊕= c; b <<< 7

- Odd round

  QR(0,4,8,12) //col 1

  QR(1,5,9,13) // col 2

  QR(2,6,10,14) // col 3

  QR(3,7,11,15) //col 4

- Even round

  QR(0,5,10,15) //diag 1

  QR(1,6,11,12) //diag 2

  QR(2,7,8,13) //diag 3

  QR(3,4,9,14) //diag 4

0 expa	1 nd 3	2 2-by	3 te k
4 Key	5 Key	6 Key	7 Key
8 Key	9 Key	10 Key	11 Key
12 Pos	13 Pos	14 None	15 Nonce

1. Use the files from the lab06/part_02 directory that you previously downloaded from repo.331.com.

2. Before you look at diffusion in ChaCha20, decrypt the message cha_ciphertext using the key Obi-Wan Kenobi and the iv of JediYoda. You will not be setting the positions at this time. The syntax will be very similar to what you did when using OpenSSL and RC4, but you will need the -iv flag in addition to the -K flag. If you need help with the command you can type `openssl enc -help` or ask the TA for hints. **Include a screenshot of your decryption command and the message in your lab report.**
3. Using the chacha20_quarter_round_skel.py code determine the diffusion between the initial matrix values and the output from the first odd quarter round (odd qr 0). You will calculate diffusion by taking the number of bits changed divided by the total number of bits (512). It will be a percentage just for this first odd quarter round.
4. Remember the values stored in the matrix are 32 bit words and printed in this code in decimal format. You will have to do a little work in the skeleton code to get the binary values and then a function to compare the bits and a final percentage of how many bits changed in the first quarter round. **Include your bit counts and the percentage of diffusion you calculated in your lab report. Upload your code to Canvas as a separate file.**

Lab 06 Template

1) Part 1:
    a. **Message encrypted with RC4**
    (5 points)

    b. **Screenshot using echo and pipe to encrypt and decrypt**
    (5 points)

    c. **Code for each of the three flaws**
    (40 points)

    d. **First byte of message and key**
    (10 total points, 5 points message, 5 points key)

2) Part 2:
    a. **Screenshot of decryption command**
    (10 points total, 5 screenshot, 5 message)

b. **Diffusion bit counts and percentage for quarter round 0**
   (10 points)

c. **Code for calculating diffusion bit counts and percentage**
   (20 points)