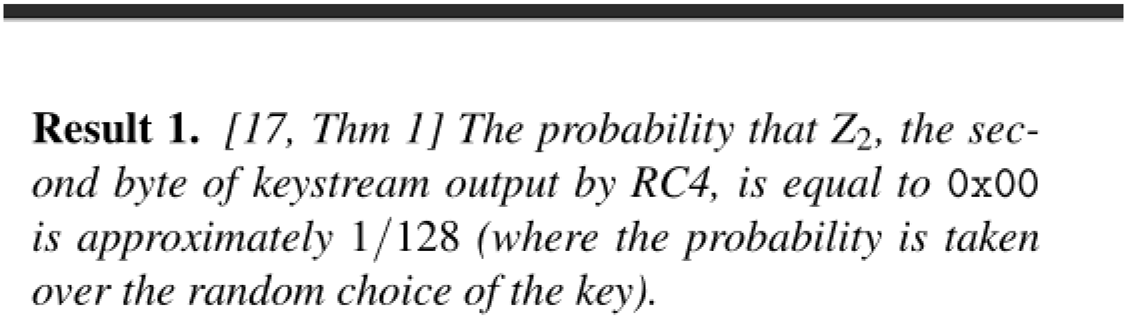
A minute attempt to break RC4 stream cipher using pre-existing biases

The program file “attempt1.ipynb” captures a minute attempt to break RC4 stream cipher using pre-existing biases. The approach that I have followed has been borrowed from the research work of Bernstein *et al.* (https://www.usenix.org/system/files/conference/usenixsecurity13/sec13-paper_alfardan.pdf).

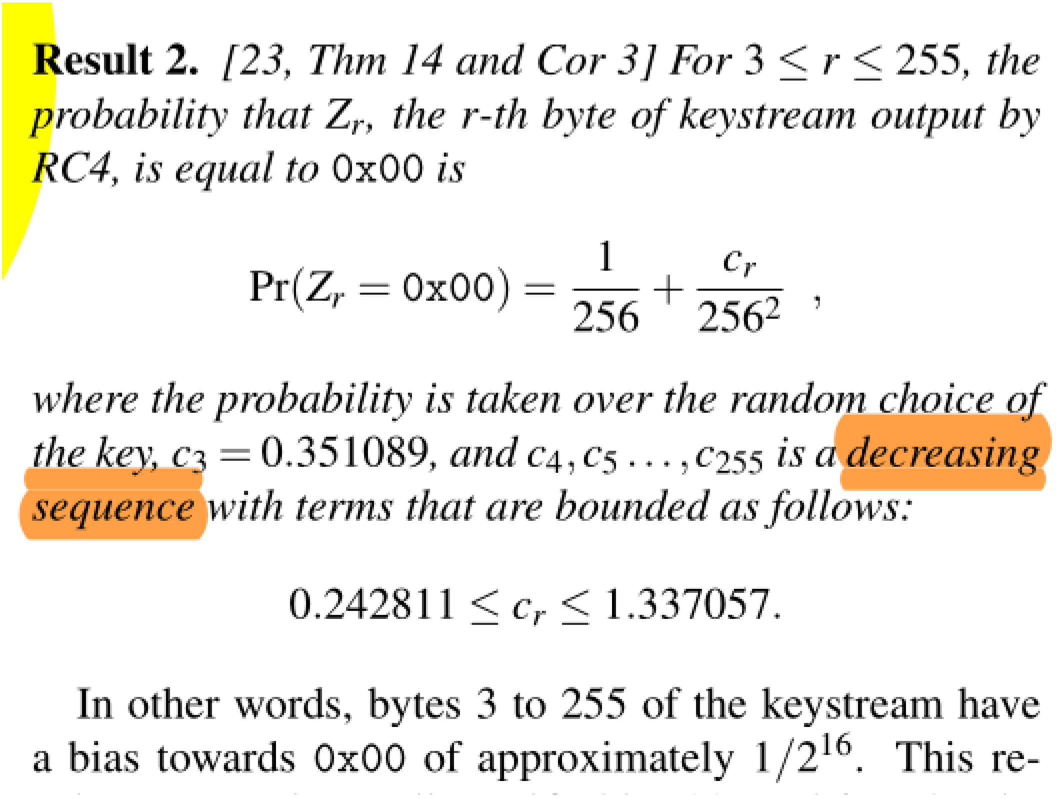
Observations:

1. There are pre-existing biases towards zero (0x00) for the second byte of a multi-byte output stream that RC4 generates.



Result 1. *[17, Thm 1] The probability that Z_2, the second byte of keystream output by RC4, is equal to* 0x00 *is approximately* 1/128 *(where the probability is taken over the random choice of the key).*

2. There are also non-negligible biases towards zero for any byte of a multi-byte output stream of RC4.



Result 2. *[23, Thm 14 and Cor 3] For $3 \leq r \leq 255$, the probability that Z_r, the r-th byte of keystream output by RC4, is equal to* `0x00` *is*

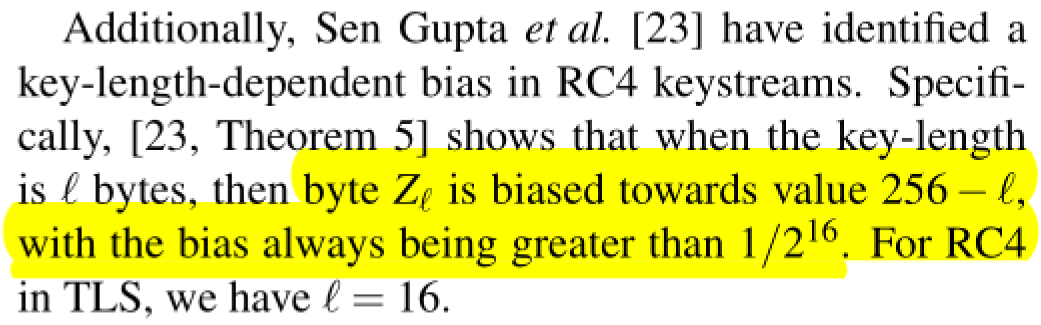
$$\Pr(Z_r = \texttt{0x00}) = \frac{1}{256} + \frac{c_r}{256^2} \quad ,$$

where the probability is taken over the random choice of the key, $c_3 = 0.351089$, and $c_4, c_5 \ldots, c_{255}$ is a decreasing sequence with terms that are bounded as follows:

$$0.242811 \leq c_r \leq 1.337057.$$

In other words, bytes 3 to 255 of the keystream have a bias towards `0x00` of approximately $1/2^{16}$. This re-

3. There are also key-length dependent biases in RC4 keystreams, first observed by Sen Gupta *et al.* (https://link.springer.com/article/10.1007/s00145-012-9138-1)



Additionally, Sen Gupta *et al.* [23] have identified a key-length-dependent bias in RC4 keystreams. Specifically, [23, Theorem 5] shows that when the key-length is ℓ bytes, then byte Z_ℓ is biased towards value $256 - \ell$, with the bias always being greater than $1/2^{16}$. For RC4 in TLS, we have $\ell = 16$.

4. There are few more biases in the keystreams which do not have a theoretical explanation.

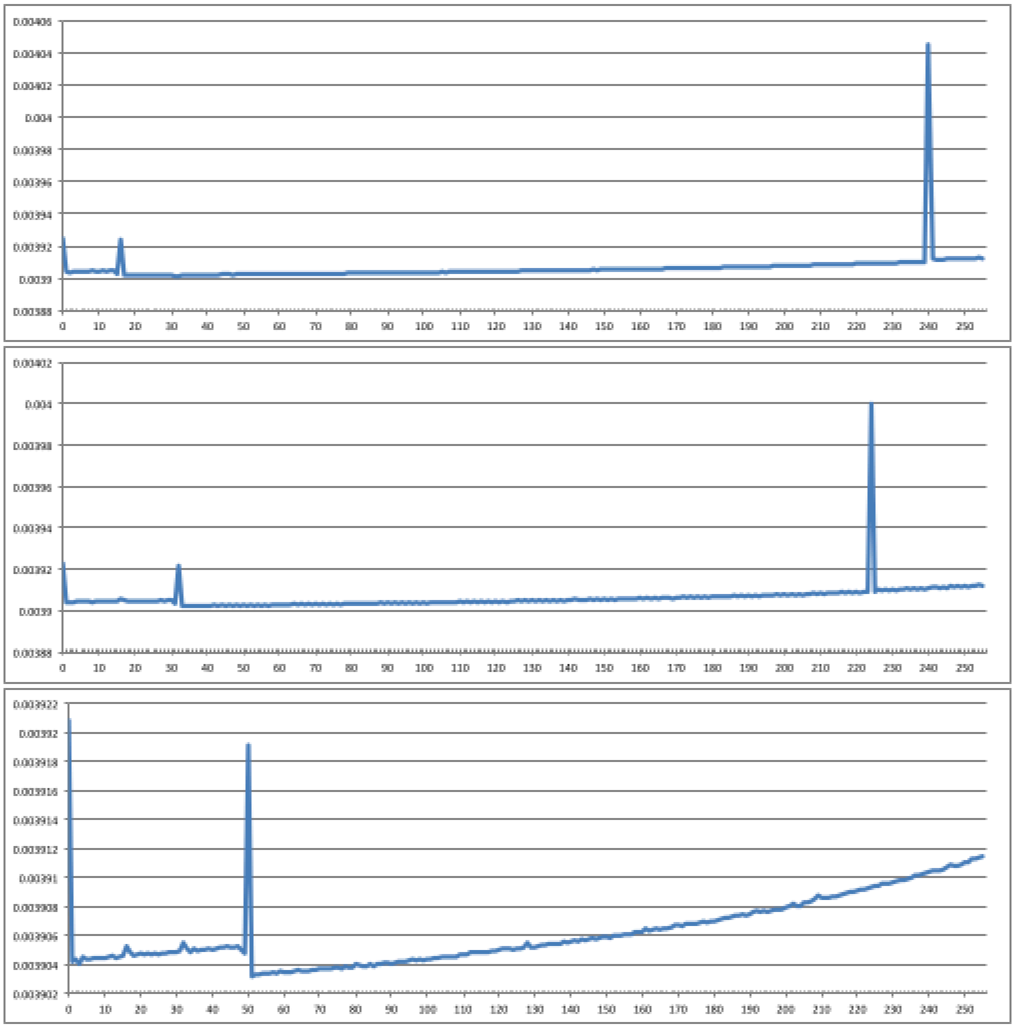
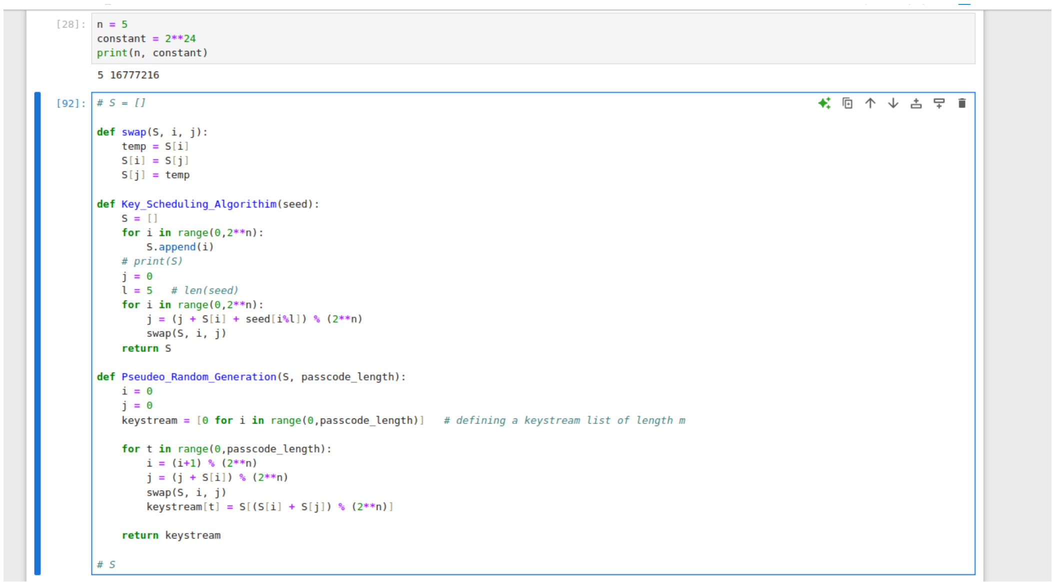


Figure 2: Measured distributions of RC4 keystream bytes Z_{16} (top), Z_{32} (middle), and Z_{50} (bottom).

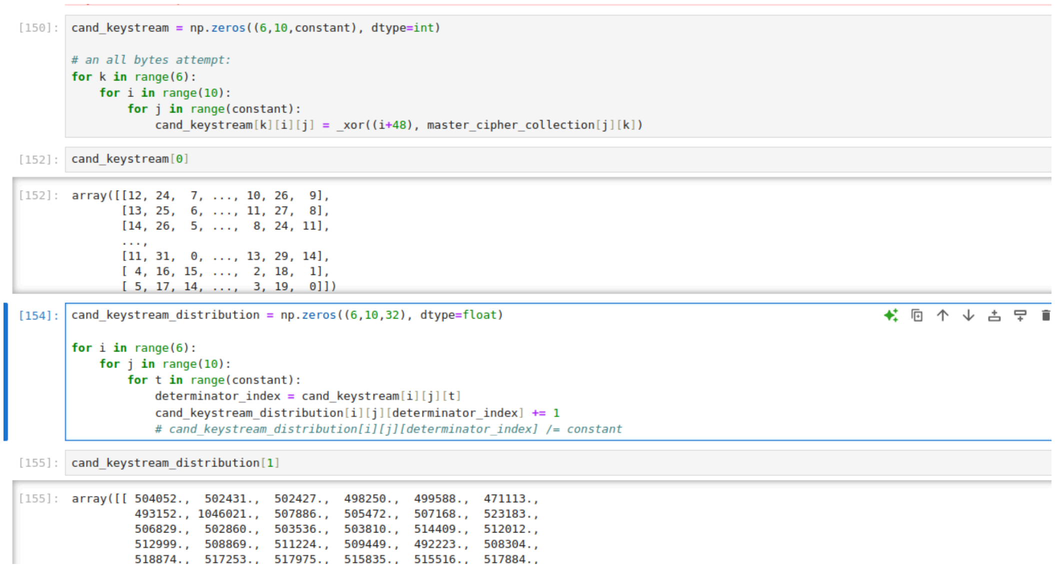
Approach:

1. Based on the above observations, I had developed an algorithm that holistically captures any existing biases in the keystream.
2. First I have defined the RC4 key-scheduling algorithm and key-generation algorithm.



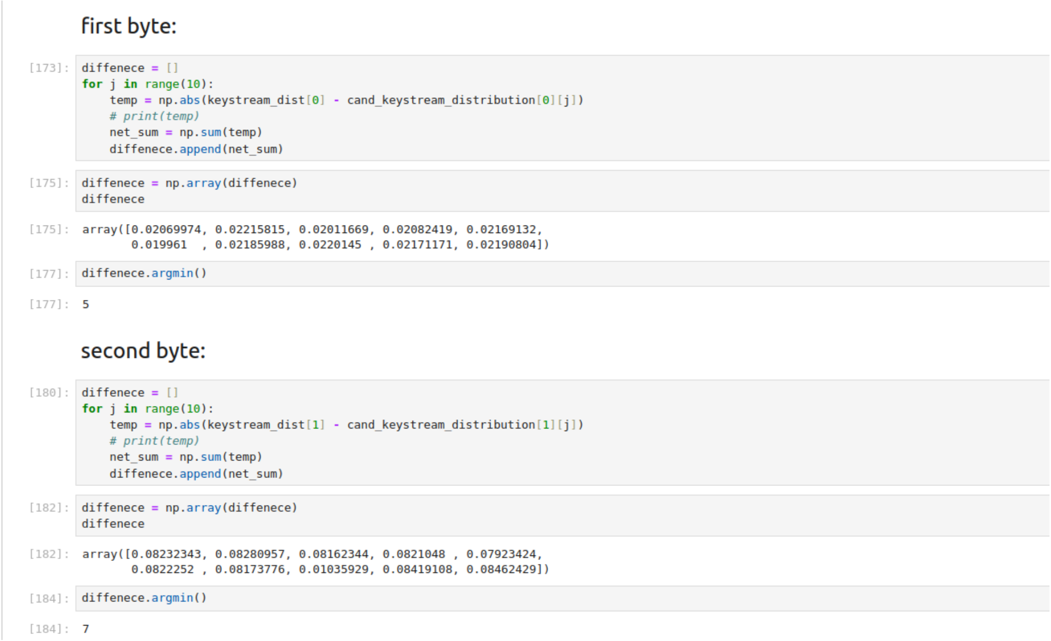
```
[28]: n = 5
      constant = 2**24
      print(n, constant)

      5 16777216

[92]: # S = []

      def swap(S, i, j):
          temp = S[i]
          S[i] = S[j]
          S[j] = temp

      def Key_Scheduling_Algorithim(seed):
          S = []
          for i in range(0,2**n):
              S.append(i)
          # print(S)
          j = 0
          l = 5   # len(seed)
          for i in range(0,2**n):
              j = (j + S[i] + seed[i%l]) % (2**n)
              swap(S, i, j)
          return S

      def Pseudeo_Random_Generation(S, passcode_length):
          i = 0
          j = 0
          keystream = [0 for i in range(0,passcode_length)]   # defining a keystream list of length m

          for t in range(0,passcode_length):
              i = (i+1) % (2**n)
              j = (j + S[i]) % (2**n)
              swap(S, i, j)
              keystream[t] = S[(S[i] + S[j]) % (2**n)]

          return keystream

      # S
```

3. Next, I have generated 2^24 such keystreams, each of length 6 bytes. Note that all of the byes will have entries 0 to 31. “All_keystream_generation” function takes care of this.
4. In “ksd_matrix” numpy nd-array, I am captureting the distribution of the key-bytes (0 to 31) for each 6 bytes.
5. Next, in a similar fashion, I am calculating the candidate keystreams from the given ciphertexts, by xor-ing them with 0 to 9 (in ascii). This candidate keystream is stored in cand_keystream 3-d matrix having shape (6, 10, 2^24).
6. In a similar fashion, I am again capturing the distribution of keystream bytes in the array “cand_keystream_distribution”.



```
[150]: cand_keystream = np.zeros((6,10,constant), dtype=int)

       # an all bytes attempt:
       for k in range(6):
           for i in range(10):
               for j in range(constant):
                   cand_keystream[k][i][j] = _xor((i+48), master_cipher_collection[j][k])

[152]: cand_keystream[0]

[152]: array([[12, 24,  7, ..., 10, 26,  9],
              [13, 25,  6, ..., 11, 27,  8],
              [14, 26,  5, ...,  8, 24, 11],
              ...,
              [11, 31,  0, ..., 13, 29, 14],
              [ 4, 16, 15, ...,  2, 18,  1],
              [ 5, 17, 14, ...,  3, 19,  0]])

[154]: cand_keystream_distribution = np.zeros((6,10,32), dtype=float)

       for i in range(6):
           for j in range(10):
               for t in range(constant):
                   determinator_index = cand_keystream[i][j][t]
                   cand_keystream_distribution[i][j][determinator_index] += 1
                   # cand_keystream_distribution[i][j][determinator_index] /= constant

[155]: cand_keystream_distribution[1]

[155]: array([[ 504052.,  502431.,  502427.,  498250.,  499588.,  471113.,
                493152., 1046021.,  507886.,  505472.,  507168.,  523183.,
                506829.,  502860.,  503536.,  503810.,  514409.,  512012.,
                512999.,  508869.,  511224.,  509449.,  492223.,  508304.,
                518874.,  517253.,  517975.,  515835.,  515516.,  517884.,
```

7. Lastly, for calculating the passcode digits, I am comparing the distributions for each passcode byte. The closest distribution should give away the most likely passcode.



first byte:

```
[173]: diffenece = []
       for j in range(10):
           temp = np.abs(keystream_dist[0] - cand_keystream_distribution[0][j])
           # print(temp)
           net_sum = np.sum(temp)
           diffenece.append(net_sum)

[175]: diffenece = np.array(diffenece)
       diffenece

[175]: array([0.02069974, 0.02215815, 0.02011669, 0.02082419, 0.02169132,
              0.019961  , 0.02185988, 0.0220145 , 0.02171171, 0.02190804])

[177]: diffenece.argmin()

[177]: 5
```

second byte:

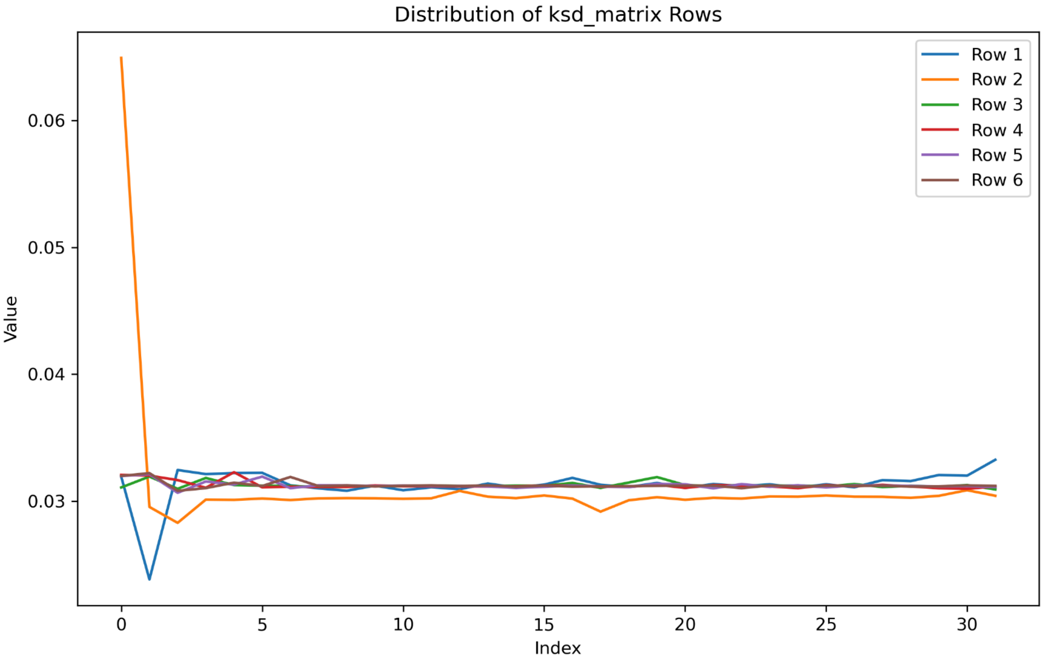
```
[180]: diffenece = []
       for j in range(10):
           temp = np.abs(keystream_dist[1] - cand_keystream_distribution[1][j])
           # print(temp)
           net_sum = np.sum(temp)
           diffenece.append(net_sum)

[182]: diffenece = np.array(diffenece)
       diffenece

[182]: array([0.08232343, 0.08280957, 0.08162344, 0.0821048 , 0.07923424,
              0.0822252 , 0.08173776, 0.01035929, 0.08419108, 0.08462429])

[184]: diffenece.argmin()

[184]: 7
```

Final result:

The final passcode that I have got: 575105. This result have been unchanged throughout multiple iterations of the experiment, even though the seed generation policy is changed.

Plotting the biases in RC4 keystreams:

Here I have plotted ksd_matrix.


Distribution of ksd_matrix Rows
Value
0.06
0.05
0.04
0.03
Index
0
5
10
15
20
25
30
Row 1
Row 2
Row 3
Row 4
Row 5
Row 6