RC4
Explanation and Cryptanalysis

Advay Bapat 23B1225
Pranav Malpani 23B1279

September 2024

Abstract

In this report, we will conduct a thorough exploration of the RC4 cipher, examining its implementation techniques and the specific algorithms imperative to its operation. We will also delve into the cryptanalysis of RC4, highlighting its vulnerabilities and the weaknesses that have been uncovered over time. Furthermore, we will analyze the various types of attacks that can be executed against the RC4 cipher, illustrating through code how these vulnerabilities can be exploited in practice.

Introduction

The stream cipher RC4, originally designed by Ron Rivest, became public in 1994 and found application in a wide variety of cryptosystems; well-known examples include SSL/TLS, WEP, WPA, and some Kerberos related encryption modes. RC4 has a remarkably short description and is extremely fast when implemented in software. However, these advantages come at the price of lowered security: several weaknesses have been identified in RC4 a few of which will be discussed and demonstrated in this paper and the code attached.
Now, we describe the TLS protocol which used RC4 and for which the attack we will show has been tested and carried out. Data to be protected by TLS is received from the application and may be fragmented and compressed before further processing. An individual record R (viewed as a sequence of bytes) is then processed as follows-

- The sender maintains an 8-byte sequence number SQN which is incremented for each record sent.
- The sender also forms a 5-byte field HDR consisting of a 2-byte version field, a 1-byte type field, and a 2-byte length field. This is nothing but a identifier of the message.
- It then calculates an `HMAC` over the string HDR||SQN||R; let T denote the resulting tag from the `HMAC`

For RC4 encryption, record and tag are concatenated to create the plaintext $P = R||T$. This plaintext is then XORed in a byte-by-byte fashion using the RC4 keystream, i.e., the ciphertext bytes are computed as

$$C_r = P_r \oplus Z_r$$

where P_r is the r^{th} plaintext byte and Z_r is the rth output of the RC4 algorithm. This RC4 algorithm is initialized with a 128 bit encryption key which we assume has already been shared between the two nodes. The actual algorithm for this is also part of TLS and is known as the TLS handshake protocol.
The initialisation of RC4 in TLS is the standard one. Notably, none of the initial keystream bytes is discarded when RC4 is used in TLS, despite these bytes having known weaknesses which we will show below.

Working

Key Scheduling Algorithm

The first component of RC4 is a Key Scheduling Algorithm, which takes in a variable-length key and converts it into a random permutation of numbers from 0 to 255. It maintains an internal state consisting of an array S of 256 bytes plus two for the pointers i and j.

- Step 1: Initialize the array to contain all numbers from 0 to 255 in order.
- Step 2: Pick ith byte from the key (the key can be variable length, so we make this operation cyclic)
- Step 3: Change the position of the pointer j, depending on the current value of S and the key value
- Step 4: Swap the ith and jth bytes. Increase pointer i and go back to step 3.

input: string of bytes s
for i $\leftarrow$ 0 to 255 do: S[i] $\leftarrow$ i
j $\leftarrow$ 0
for i $\leftarrow$ 0 to 255 do:
$k \leftarrow s[i \mod |s|]$ # extract one byte from seed
$j \leftarrow (j + S[i] + k) \mod 256$
$swap(S[i], S[j])$

For the attacks that we discuss, this permutation is assumed to be random. It is important to note that this algorithm works in constant time.

Pseudorandom generation Algorithm

Now, we shall generate the keystream using the "PRG" algorithm. The algorithm takes as input the output of KSA, the random permutation, and keeps updating the array S by using the swap operation.

- Initialize i and j to 0
- increment i by 1, and j by $(j + S[i])$

- swap the ith and jth bytes
- output the byte at the position given by the ith byte + the jth byte

$i \leftarrow 0$, $j \leftarrow 0$
repeat
$i \leftarrow (i + 1) \mod 256$
$j \leftarrow (j + S[i]) \mod 256$
$swap(S[i], S[j])$
output $S[(S[i] + S[j]) \mod 256]$
forever

After generating the key, the cipher follows the same steps as that of One Time Pad, where the key and the message are XORed. As we can see, this algorithm does not stop by itself, it is upto Alice and Bob to stop it. Usually, the output was only taken as long as the input and the algorithm stopped beyond that point. We shall see how this makes it an insecure cipher.

Biases in RC4

At one point RC4 was believed to be a secure stream cipher and was widely deployed in applications.The cipher fell from grace after a number of attacks showed that its output is somewhat biased. We present the Mantin-Shamir Bias, the Single Byte Bias and Multi-byte bias.

Mantin-Shamir Bias

The RC4 setup algorithm initializes the array S to a permutation of 0..255 generated from the given random seed. For now, let us assume that the RC4 setup algorithm is perfect and generates a uniform permutation from the set of all 256! permutations. Mantin and Shamir showed that, even assuming perfect initialization, the output of RC4 is biased.
The following is the lemma for the bias-
Suppose the array S is set to random permutation of $0...n-1$ and that i, j are set to 0. Then the probability that the second byte of the output of RC4 is equal to 0 is $2/n$.
Proof: We define the event P such that $S[2] = 0$ and $S[1] \neq 2$. As we can see from the Figure 1 if this event occurs, the probability of the we getting $Z_2 = 0$

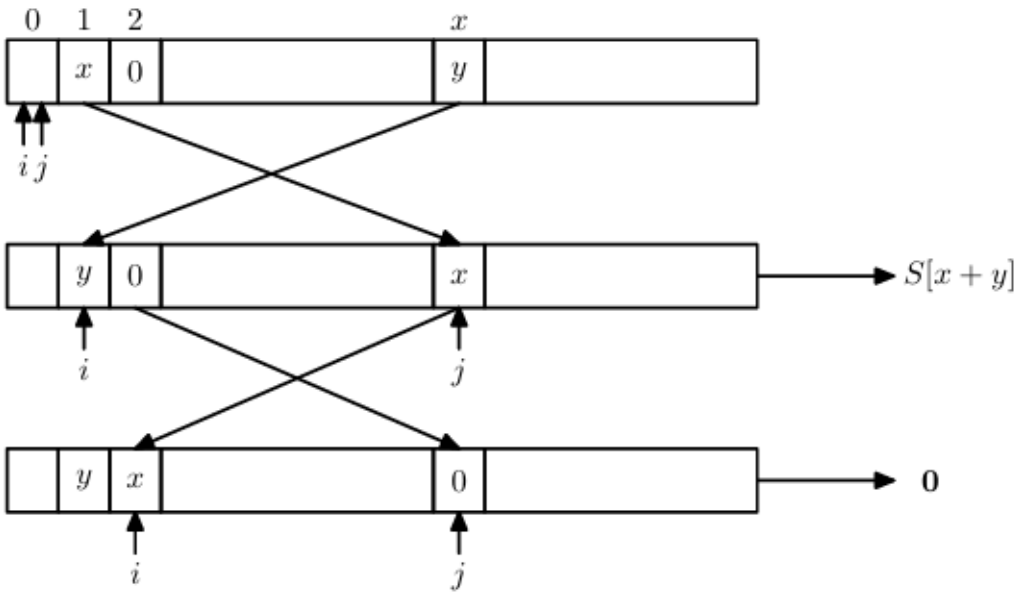


Figure 1: Proof of lemma

is 1. This is because for any x, the first iteration of RC4 algorithm produces the state where x is at the $x^{th} = j^{th}$ position. This produces $S[S[i] + S[j]]$ which is nothing but $S[x + y]$. The second iteration then makes $i = 2$, and since $S[i] = 0$, j is not updated and remains equal to x with $S[x] = S[j] = x$ .Thus, after swaps we have $S[x] = 0$ and $S[i] = x$. Finally, when producing the second output we look at $S[S[x] + S[i]]$ which is nothing but $S[x] = 0$. We can see here that if $S[1] = 2$, we end up replacing 0 with 2 and in the next step we get $j = 4$ which clearly doesn't ensure $Z_2 = 0$. Thus we can write the probability of $Z_2 = 0$ as-

$$Pr(Z_2 = 0) = Pr(Z_2 = 0|P)Pr(P) + Pr(Z_2 = 0|\bar{P})Pr(\bar{P})$$

$$Pr(Z_2 = 0) = (1)\big(\frac{1}{n}\big)\big(\frac{n-2}{n-1}\big) + \big(\frac{1}{n}\big)\big(1 - \frac{1}{n}\big)\big(\frac{n-2}{n-1}\big)$$

For large enough n ($n = 256$)-

$$Pr(Z_2 = 0) \approx \frac{2}{n}$$

This leads to a simple distinguisher for the RC4 PRG. Given a string $x \in \{0...255\}^l$, for $l \geq 2$, the distinguisher outputs 0 if the second byte of x is 0 and outputs 1 otherwise. By the lemma this distinguisher has advantage approximately $1/n$, which is 0.39% for RC4. But, this is not sufficient to construct a full-fledged plaintext recovery attack. As it turns out, RC4 has numerous other biases in different places at the keystream. We explore them next.

Single-Byte Biases

There are many more baises to be found in RC4 keystream. In moving towards the overall single-byte biases, we first explore some of the individual single byte biases. One of these biases was obtained by Sen Gupta et al.as a refinement of an earlier result of Maitra et al:
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

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

where the probability is taken over the random choice of the key, $c_3 = 0.351089$, and $c_4, c_5..., c_{255}$ 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 $1/2^{16}$.
There are a few more biases towards certain ouptuts of the RC4 keystream. These have been mostly found empirically, the proofs for some of the biases do exist but they are quite involved as far as we know, so we just present the biases themselves.
For Z_{16}, we have 3 main biases: the bias towards `0x00`, the very dominant key-length-dependent bias towards `0xF0` (decimal 240 which is 256-16) from, and a new bias towards `0x10` (decimal 16). Similarly for Z_{32}, we also have 3 main biases: the bias towards `0x00`, a large, new bias towards `0xE0` (decimal 224 which is 256-32), and a new bias towards `0x20` (decimal 32).
For Z_{50}, there are significant biases towards byte values `0x00` and `0x32` (decimal 50), as well as an upward trend in probability as the byte value increases. All of these biases can be seen in the figures below.

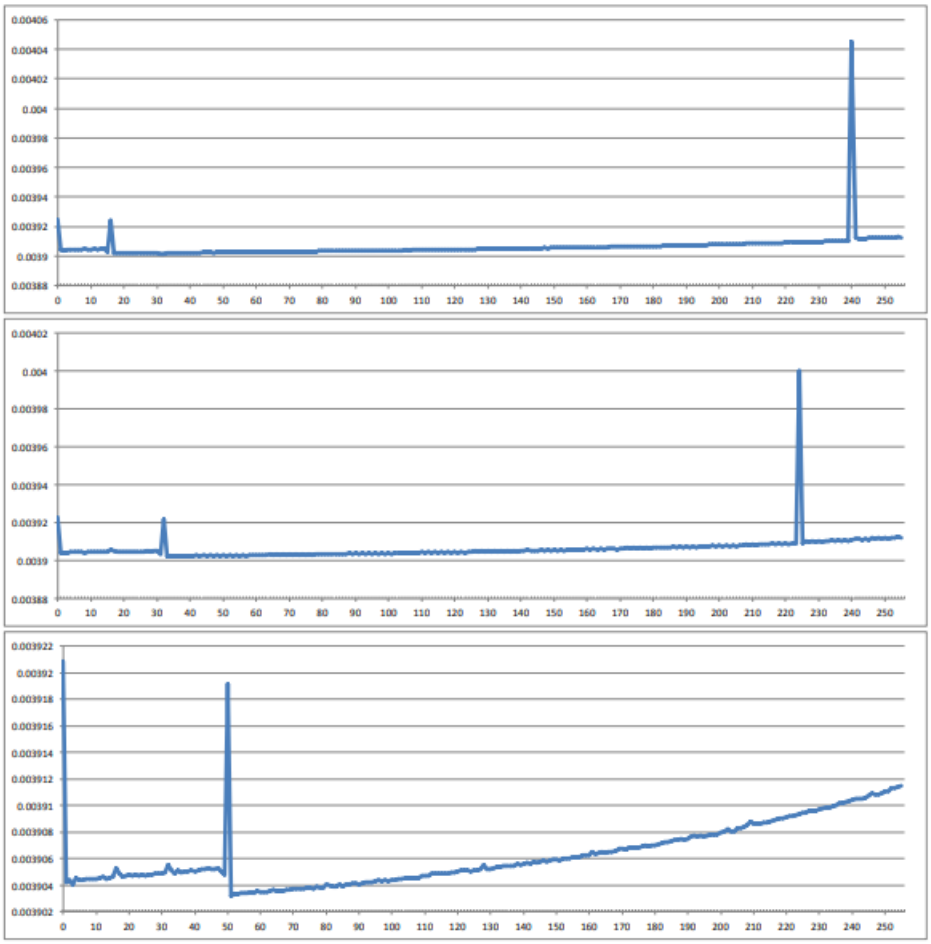


Figure 2: Measured distributions of RC4 keystream bytes Z16 (top), Z32 (middle), and Z50 (bottom). X axis is the number which might appear in the keystream at that position while the Y axis is scaled and represents probability of that number appearing.

The existence of these biases prevents a simplistic attack using just the 0 biases at different positions. The biases in some sense protect each other from being exploited individually. Thus, we now move towards the single byte bias attack which uses all of the biases together to recover the first 256 bytes of the keystream.

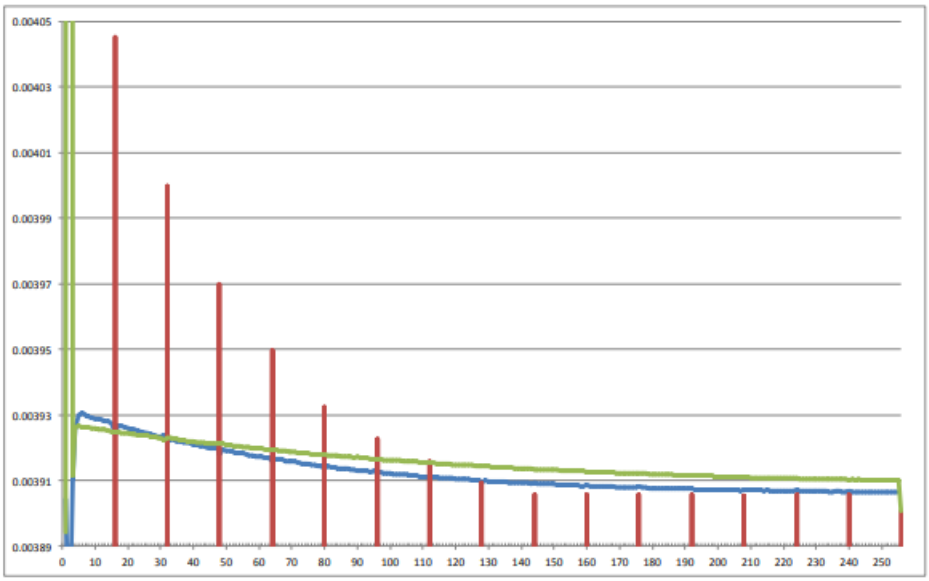


Figure 3: Measured strength of the bias towards 0x00 (green), the bias towards value r in Z_r (blue), and the keylength dependent bias towards byte value $256 - r$ (red) for keystream bytes $Z_1, ..., Z_{256}$. Note that the large peak for the 0x00 bias in Z_2 extends beyond the bounds of the graph and is not fully shown, this only illustrates how large the Mantin-Shamir bias is

Multiple-Byte Biases

In addition to the single-byte biases previously mentioned, various multi-byte biases have also been identified in the RC4 keystream. Many of these multi-byte biases appear at regular intervals within the keystream. Fluhrer and McGrew conducted a detailed analysis of the distribution of consecutive keystream byte pairs (Z_r, Z_{r+1}), uncovering a wide range of multi-byte biases. Their study involved scaled-down versions of RC4, operating under the assumption of an idealized internal state where both the permutation S and the internal variable j behave randomly. They then extrapolated their findings to standard RC4.

Al Fardan *et al* in 2013 experimentally verified the Fluhrer-McGrew biases by analysing the output of 2^{10} RC4 instances using 128-bit keys and generating 240 keystream bytes each. For each keystream, the initial 1024 bytes

Byte pair	Condition on i	Probability
$(0,0)$	$i=1$	$2^{-16}(1+2^{-9})$
$(0,0)$	$i\neq 1,255$	$2^{-16}(1+2^{-8})$
$(0,1)$	$i\neq 0,1$	$2^{-16}(1+2^{-8})$
$(i+1,255)$	$i\neq 254$	$2^{-16}(1+2^{-8})$
$(255,i+1)$	$i\neq 1,254$	$2^{-16}(1+2^{-8})$
$(255,i+2)$	$i\neq 0,253,254,255$	$2^{-16}(1+2^{-8})$
$(255,0)$	$i=254$	$2^{-16}(1+2^{-8})$
$(255,1)$	$i=255$	$2^{-16}(1+2^{-8})$
$(255,2)$	$i=0,1$	$2^{-16}(1+2^{-8})$
$(129,129)$	$i=2$	$2^{-16}(1+2^{-8})$
$(255,255)$	$i\neq 254$	$2^{-16}(1-2^{-8})$
$(0,i+1)$	$i\neq 0,255$	$2^{-16}(1-2^{-8})$

Figure 4: Findings of Fluhrer and McGrew

were dropped. Based on this data, they found the biases from Fluhrer and McGrew to be accurate, also for 128-bit keys.

Attack

We explain the Single-Byte Bias Attack as implemented by Al Fardan et al, and then show our own small-scale implementation.

Al Fardan et al

The idea is to first obtain a detailed picture of the distributions of RC4 keystream bytes Z_r, for all positions r, by gathering statistics from keystreams generated using a large number of independent keys (2^{44} in the case of this attack). That is, we obtain $P_{r,k}$ for $r \in$`{0,...256}` and $k \in$`{0x00,...0xFF}`, where r is the position in the keystream and k is the possible value of the element.
Having obtained the probability distribution of k over all positions r, we now move towards making the testcases for our attack. We choose a fixed plaintext m and generate S ciphertexts from this message where each ciphertext has a different key. This gives us S number of sample bytes per position.
Next, we guess the plaintext value m at each position totalling in 256^2 guesses

at max. The guessed plaintext value is XORed with the S number of ciphertexts creating a pseudo-distribution of keys at that position. This pseudo distribution is stored and the plaintext giving the closest distribution to the one found empirically is finalised as the plaintext at that position.
For different values of S we get varying levels of accuracy.

- Even with as few as 2^{24} sessions, some positions of the plaintext are correctly recovered with high probability. The ones with highest probability seem to arise because of the key-length-dependent biases that we observed in positions that are multiples of 16. These large biases make it easier to recover the correct plaintext bytes when compared to other ciphertext positions.

- With $S = 2^{26}$ sessions, the first 46 plaintext bytes are recovered with rate at least 50% per byte.

- The rate at which bytes are correctly recovered increases steadily as the number of sessions S is increased, with all but the last few bytes being reliably recovered already for 2^{31} trials.

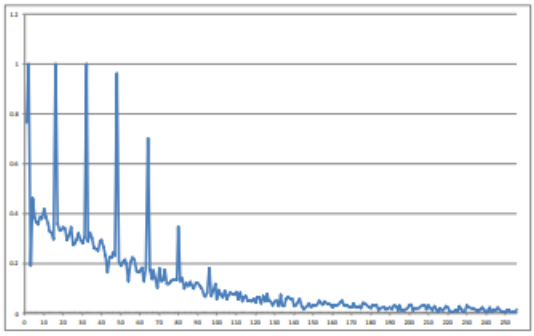


Figure 4: Recovery rate of the single-byte bias attack for $S = 2^{24}$ sessions for first 256 bytes of plaintext (based on 256 experiments).

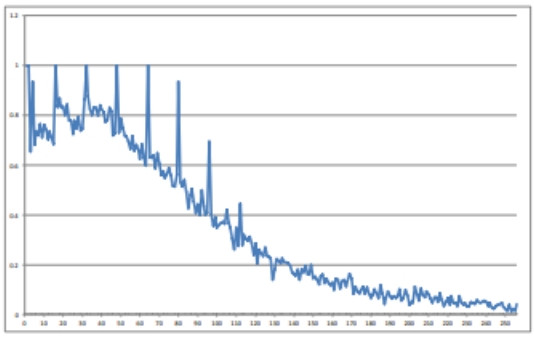


Figure 5: Recovery rate of the single-byte bias attack for $S = 2^{26}$ sessions for the first 256 bytes of plaintext (based on 256 experiments).

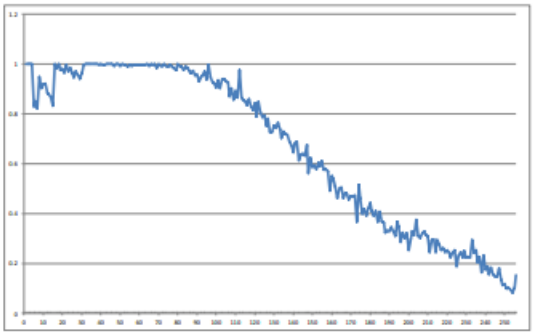


Figure 6: Recovery rate of the single-byte bias attack for $S = 2^{28}$ sessions for the first 256 bytes of plaintext (based on 256 experiments).

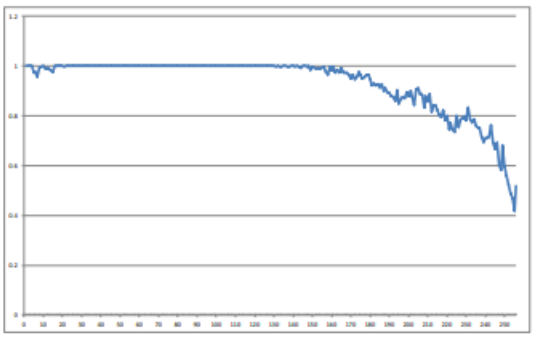


Figure 7: Recovery rate of the single-byte bias attack for $S = 2^{30}$ sessions for the first 256 bytes of plaintext (based on 256 experiments).

Figure 5: The figures obtained by Al Fardan et al

Our Implementation

We constructed an attack much like the one described above, except on a smaller scale, redefining a byte to be 3-bits long, and used 8-byte (24 bit) keys. We generate 8^8 keystreams (considering only the first 8 bytes), using 8^8 unique seed strings. We then stored them all in a file (to save computational complexity), and for easy readout later.

Code

First, we write the code for generating all the "pseudorandom" keys.

```
def ksa(key):
    key_length = len(key)
    S = list(range(8))
    p = list(key)
    key = [int(i) for i in p]
    j = 0
    for i in range(8):
        j = (j + S[i] + key[i % key_length]) % 8
        S[i], S[j] = S[j], S[i]

    return S


def prga(S, plaintext_length):
    i = 0
    j = 0
    keystream = []

    for step in range(plaintext_length):
        i = (i + 1) % 8
        j = (j + S[i]) % 8
        S[i], S[j] = S[j], S[i]
        K = S[(S[i] + S[j]) % 8]
        keystream.append(K)

    return keystream
```

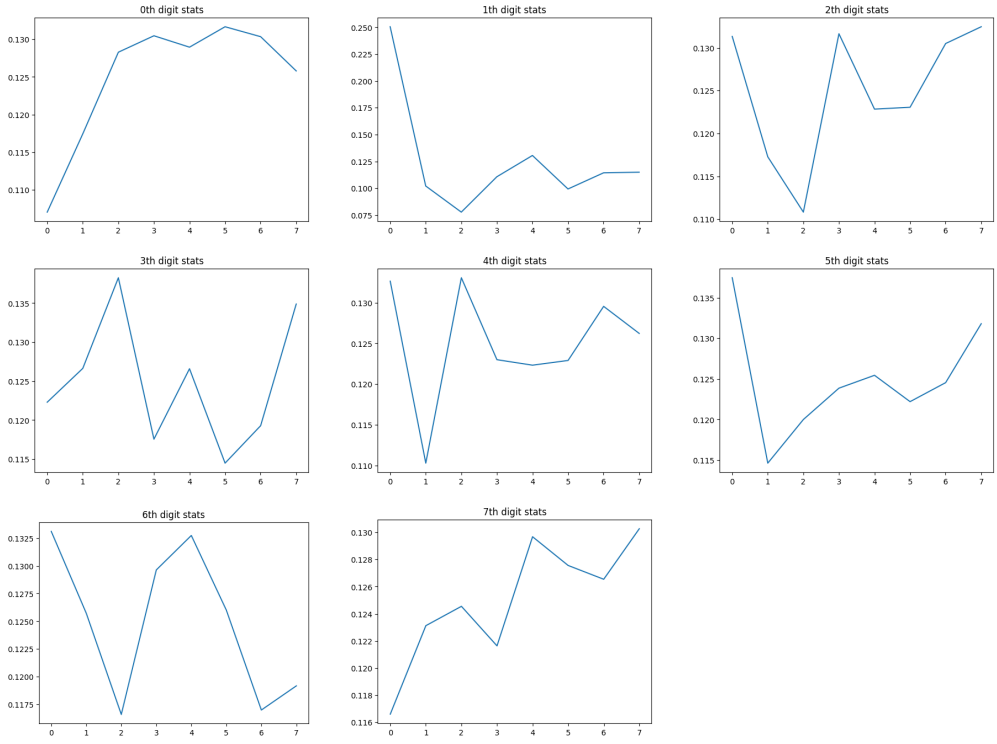
This code is run 8^8 times, in order to generate the file with all the keystreams. Now, we find the fractions of each value at each byte. This is in the format of an 8×8 matrix, the i, jth value of which corresponds to the frequency of the number j at the byte position i.

```
r = open('output-3.txt', 'r+')
import numpy as np

matrix = np.zeros((8,8))
for x in r:
    x = x.removesuffix("\n")
    for i in range(len(x)):
        matrix[i][int(x[i])] += 1

matrix = matrix / (8**8)
print(matrix)
```

This gives us the matrix

$$\begin{bmatrix}
0.1070413 & 0.1174373 & 0.12827766 & 0.13046104 & 0.12894565 & 0.13165808 & 0.13033915 & 0.12579095 \\
0.25051308 & 0.10204422 & 0.07769263 & 0.11067009 & 0.13050067 & 0.09926736 & 0.11433989 & 0.11492318 \\
0.13132864 & 0.11728185 & 0.11082745 & 0.13164181 & 0.12284732 & 0.12306422 & 0.1305005 & 0.13245934 \\
0.12229335 & 0.12662894 & 0.13824922 & 0.11755455 & 0.12658405 & 0.11448288 & 0.1192826 & 0.13487554 \\
0.13262612 & 0.1103034 & 0.13304937 & 0.12299496 & 0.12232107 & 0.12290102 & 0.12953895 & 0.12621623 \\
0.13746125 & 0.11463201 & 0.12000132 & 0.1238625 & 0.12544799 & 0.1222024 & 0.12454438 & 0.13179928 \\
0.13309634 & 0.12571895 & 0.11659956 & 0.12962526 & 0.13272959 & 0.12600243 & 0.11699969 & 0.11917931 \\
0.11659908 & 0.12312132 & 0.12454855 & 0.12163216 & 0.12967831 & 0.12756079 & 0.1265431 & 0.1302678
\end{bmatrix}$$

Here is the frequency distribution for all the byte biases:



Thus, we can empirically verify the Mantin-Shamir bias, by seeing that the probability for 0 in the 2^{nd} bit is double what it should be, i.e. 0.25 instead of 0.125.
Now, we select a random plaintext to encode (we pick the message length to be 8 bits long, since we generated keys only upto that point, but in general this can be extended to more bits), print the plaintext as is, and then encode it.

```
pt = np.random.randint(1, 8**8+1)
plaintext = oct(pt).split('o')[1]
pt = '0' * (8-len(plaintext))
plaintext = pt + plaintext
print(plaintext)
no_of_ciphertexts = 4000
keys = np.random.choice(8**8, no_of_ciphertexts)
keys = np.sort(keys)
ciphertexts = []
with open('output-3.txt', 'r') as r:
    for i, line in enumerate(r):
        if i == keys[0]:
            line = line.removesuffix("\n")
            n1 = int("0o"+line, 8)
            n2 = int("0o"+plaintext, 8)
            ct = oct(np.bitwise_xor(n2, n1)).split('o')[1]
            y = '0' * (8-len(ct))
            ct = y+ct
            ciphertexts.append(ct)
            keys = keys[keys!=i]
        if len(keys) == 0:
            break
```

20405306

This is our plaintext. We shall now try to recover the plaintext, by selecting 4000 keys from our initial list, and then comparing the probability distribution we receive for different guesses. The comparison method we use, is the **Cosine Similarity**, in which we convert the probability distributions into 8-dimensional vector, and calculate the $\cos\theta$ between the two vectors. We then select the guess with the highest $\cos\theta$.

```
from numpy import linalg as la
correlation = 0
final_ans = ""
for r in range(8):
    ans = 0
    correlation = 0
    for u in range(8):
        k_set = np.zeros(8)
        for j in range(len(ciphertexts)):
            cjr = ciphertexts[j][r]
            k = np.bitwise_xor(int(cjr), int(u))
            k_set[k] += 1
        k_set = k_set / no_of_ciphertexts
        row1 = matrix[r]
        row1_norm = la.norm(row1)
        row2 = k_set
        row2_norm = la.norm(row2)
        new_correl = np.correlate(row1, row2)/(row1_norm *
   row2_norm)
        if(new_correl > correlation):
            ans = u
            correlation = new_correl
    print(ans)
    final_ans = final_ans + str(ans)
print(final_ans)
```

```
2
0
4
0
5
3
0
6
20405306
```

Thus, we have recovered the plaintext with 100% accuracy.

WEP and the FMS Attack

- The IEEE 802.11b standard ratified in 1999 defines a protocol for short range wireless communication (WiFi). Security is provided by a Wired Equivalent Privacy (WEP) encapsulation of 802.11b data frames.
- When WEP is enabled, all members of the wireless network share a long term secret key k. The standard supports either 40-bit keys or 128-bit keys.
- This key is prepended with a IV and passed to RC4 as the key.
- Since this creates related keys which RC4 was not designed to handle, it becomes the victim of the related keys attack described by Fluhrer, Mantin, and Shamir aka FMS Attack.

Sources

- Al Fardan et al
- Boneh and Shoup
- The code can be found here