Stream ciphers: RC4 and others

Tanja Lange

Eindhoven University of Technology

2WF80: Introduction to Cryptology

RC4

- Designed by Ron Rivest. In use since 1987.
- Very simple description, efficient to implement in software.
- Key defined as list of `l` bytes, i.e., `l` integers in $[0, 255]$.
  Minimum `l` $= 5$, so 2^{40} cost of brute-force attacks.
  Maximum `l` $= 256$, but typically no more than 16 bytes.
- Cipher uses a length-256 state vector `S` containing a permutation of $\{0, 1, 2, 3, \ldots, 255\}$, starting with `S[i] = i`.

```
# feed in the key, key has length l
j = 0
for i = 0 to 255:
    j = (j + S[i] + key[i mod l]) mod 256
    swap(S[i],S[j])
# generate n bytes of output stream
i = 0; j = 0
for t = 0 to n-1:
   i = (i + 1) mod 256
   j = (j + S[i]) mod 256
   swap(S[i],S[j])
   append S[(S[i] + S[j]) mod 256] to output
```

Example

Starting state of S

```
 0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15 161  17  18  19
20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  .   .   .  255
```

```
# feed in the key, key has length l
j = 0
for i = 0 to 255:
    j = (j + S[i] + key[i mod l]) mod 256
    swap(S[i],S[j])
```

Use key [10, 20, 30, 40, 50]
First 3 updates:

Example

Starting state of S

```
 0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15 161  17  18  19
20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  .   .   .  255
```

```
# feed in the key, key has length l
j = 0
for i = 0 to 255:
    j = (j + S[i] + key[i mod l]) mod 256
    swap(S[i],S[j])
```

Use key [10, 20, 30, 40, 50]

First 3 updates:

```
10   1   2   3   4   5   6   7   8   9   0  11  12  13  14  15 161  17  18  19
20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  .   .   .  255
```

Example

Starting state of S

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 161 17 18 19
20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35  .   .  .  255
```

```
# feed in the key, key has length l
j = 0
for i = 0 to 255:
    j = (j + S[i] + key[i mod l]) mod 256
    swap(S[i],S[j])
```

Use key [10, 20, 30, 40, 50]

First 3 updates:

```
10  1  2  3  4  5  6  7  8  9  0 11 12 13 14 15 161 17 18 19
20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35  .   .  .  255
```

```
10 31  2  3  4  5  6  7  8  9  0 11 12 13 14 15 161 17 18 19
20 21 22 23 24 25 26 27 28 29 30  1 32 33 34 35  .   .  .  255
```

Example

Starting state of S

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 161 17 18 19
20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 .   .  .  255
```

```
# feed in the key, key has length l
j = 0
for i = 0 to 255:
    j = (j + S[i] + key[i mod l]) mod 256
    swap(S[i],S[j])
```

Use key [10, 20, 30, 40, 50]

First 3 updates:

```
10  1  2  3  4  5  6  7  8  9  0 11 12 13 14 15 161 17 18 19
20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 .   .  .  255

10 31  2  3  4  5  6  7  8  9  0 11 12 13 14 15 161 17 18 19
20 21 22 23 24 25 26 27 28 29 30  1 32 33 34 35 .   .  .  255

10 31 63  3  4  5  6  7  8  9  0 11 12 13 14 15 161 17 18 19
20 21 22 23 24 25 26 27 28 29 30  1 32 33 34 35 .   .  .  255
```

Example

State after feeding in the key:

```
 10  31  63 106 237 132 201 238  30  89  78 130  18 144  36  58
187 141  38  65  42  83 135  25  94  64 190   0  96  54 185  84
146 200 231  21 129 196 118 230 157  76   1   4   5  79  49 115
140 199   3   7 233 126  68  80 177  90   6 151 180 202 113  86
156 172 105  61 174 138 236  71  69 248 149 143 197  93 150 166
229  17  44 119  98 137 165  97  11 213  32 168 222 167 169 211
 57 108  19 131 120 109  66 128  87  37  12 102 182  34  35 114
227  46 226 154 242  20 170 247 127  56  48  77 101 254 179 210
 67 183 204 145 175 153  13 136 235 250  50  23 195 232 110  15
155  91 221 205 134 112 234 111  72 178 194 225  14 171 218 152
162 206  95 173  47  81 193  26 142  52  28 122 125 181 251 189
 88 191 103  70 121  29 133 184  33 209 239  24 215 217 104 223
186 139  22 203 241 158 216 207 252 219 243 164  27  85 220 117
160  55 161   2 116  39 249  41 176 192 100   9 224 124  62 255
 75  16 198   8 147  82 245 188 212  40  99 240 214 208  74  43
246 244  60  53  92 107 228  73 123 253  45 159  51 148 163  59
```

First 4 bytes still in place from first 4 steps, 5th got swapped again.

Generate output and update

Each output step generates 1 byte of output and updates the state:

```
# generate n bytes of output stream
i = 0; j = 0
for t = 0 to n-1:
   i = (i + 1) mod 256
   j = (j + S[i]) mod 256
   swap(S[i],S[j])
   append S[(S[i] + S[j]) mod 256] to output
```

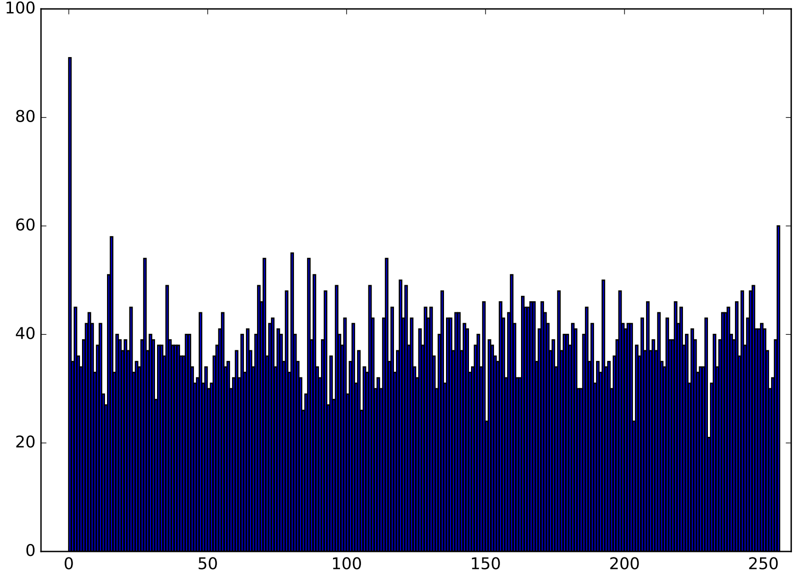
The state vector S gets updated by swaps, so continues to be a permutation of $\{0, 1, 2, 3, \ldots, 255\}$.

Note that the addition modulo 256 is on the index of the output byte, not on the values held in the positions.
Our example outputs

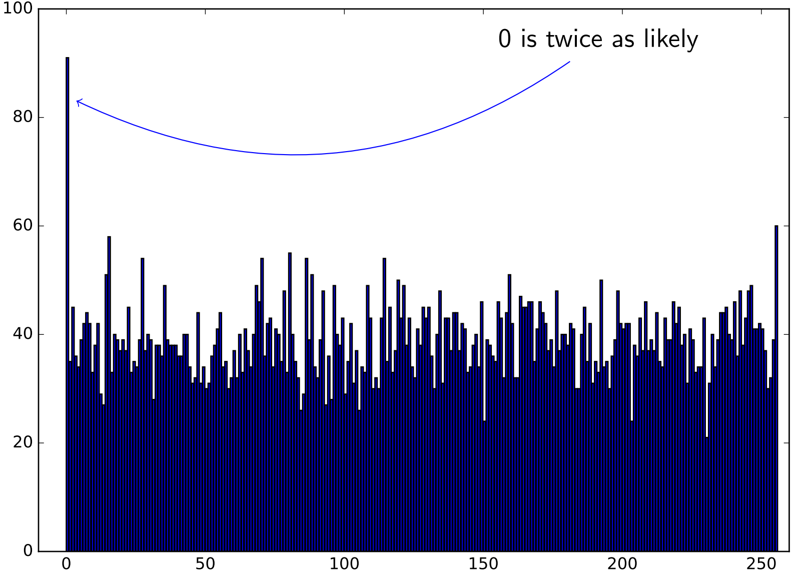
```
154, 212, 66, 78, 62, 226, 147, 105, 192, 151, 161, 237,
229, 89, 84, 91, 158, 104, 195, 25, 45, 190, 181 ...
```

To encrypt with the RC4 stream cipher, xor (add modulo 2) the message and the output (representing each byte as 8 bits).

Plotting the second output byte, 100 000 runs



Plotting the second output byte, 100 000 runs



0 is twice as likely

Why is the second byte biased towards 0?

Assume `S[2]= 0` at the end of the key setup.

Then `i = 0, j = 0`

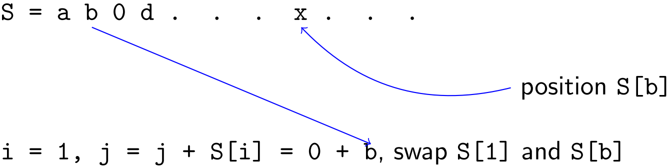
`S = a b 0 d .   .   .   x .   .   .`

position `S[b]`

Why is the second byte biased towards 0?

Assume `S[2]= 0` at the end of the key setup.

Then `i = 0, j = 0`



`S = a b 0 d .   .   .   x .   .   .`

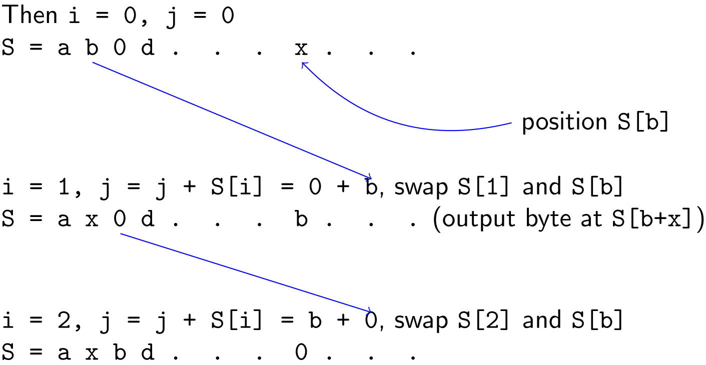
position `S[b]`

`i = 1, j = j + S[i] = 0 + b`, swap `S[1]` and `S[b]`

`S = a x 0 d .   .   .   b .   .   .` (output byte at `S[b+x]`)

Why is the second byte biased towards 0?

Assume `S[2]= 0` at the end of the key setup.



Then `i = 0, j = 0`

`S = a b 0 d .   .   .   x .   .   .`

position `S[b]`

`i = 1, j = j + S[i] = 0 + b`, swap `S[1]` and `S[b]`

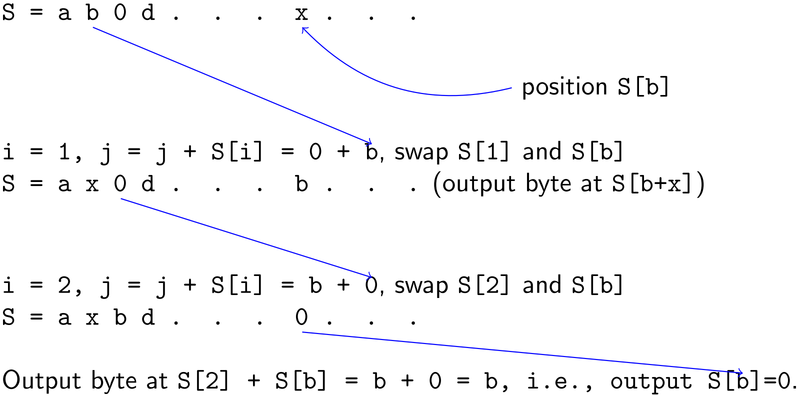
`S = a x 0 d .   .   .   b .   .   .` (output byte at `S[b+x]`)

`i = 2, j = j + S[i] = b + 0`, swap `S[2]` and `S[b]`

`S = a x b d .   .   .   0 .   .   .`

Why is the second byte biased towards 0?

Assume `S[2]= 0` at the end of the key setup.
Then `i = 0, j = 0`



`S = a b 0 d .   .   .   x .   .   .`

position `S[b]`

`i = 1, j = j + S[i] = 0 + b`, swap `S[1]` and `S[b]`
`S = a x 0 d .   .   .   b .   .   .` (output byte at `S[b+x]`)

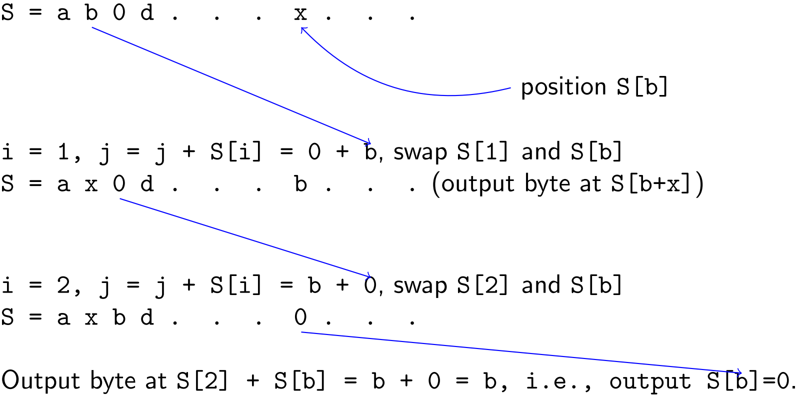
`i = 2, j = j + S[i] = b + 0`, swap `S[2]` and `S[b]`
`S = a x b d .   .   .   0 .   .   .`

Output byte at `S[2] + S[b] = b + 0 = b, i.e., output S[b]`=0.

Why is the second byte biased towards 0?

Assume `S[2]= 0` at the end of the key setup.

Then `i = 0, j = 0`



`S = a b 0 d .   .   .   x .   .   .`

position `S[b]`

`i = 1, j = j + S[i] = 0 + b`, swap `S[1]` and `S[b]`

`S = a x 0 d .   .   .   b .   .   .` (output byte at `S[b+x]`)

`i = 2, j = j + S[i] = b + 0`, swap `S[2]` and `S[b]`

`S = a x b d .   .   .   0 .   .   .`

Output byte at `S[2] + S[b] = b + 0 = b, i.e., output S[b]`=0.

This fails only if `b = 2`, else guaranteed to output 0.

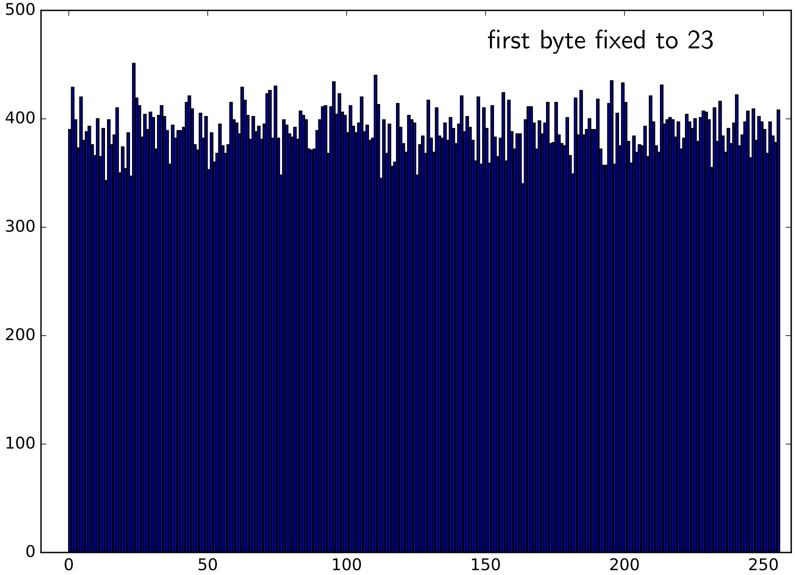
Probability of outputting 0

- Starting state with S[2]=0 happens with probability $1/256$.
  This outputs 0 unless b = 2, thus with probability $254/255$.
- No other strong biases – so for any other starting state the probability to output some value v is $1/256$,
  for a total of $255/(256)^2$ (plus a tiny bit for S[2]=0, S[1]=2).

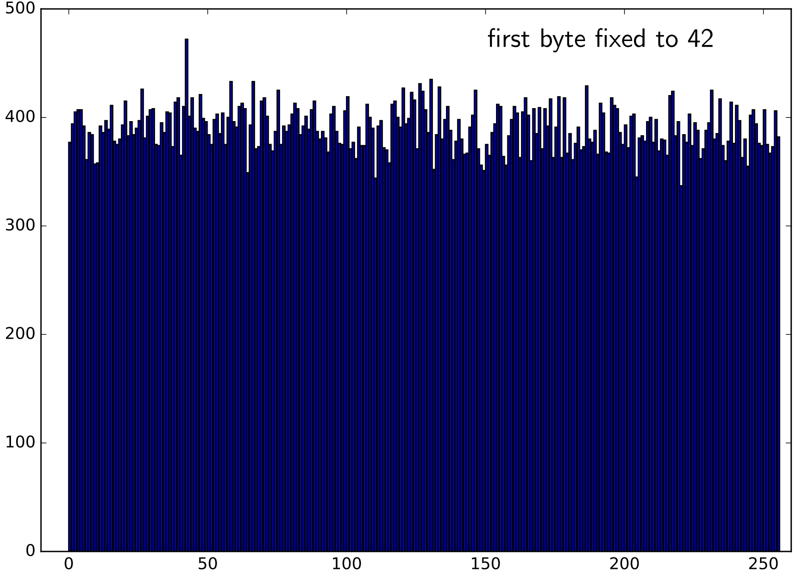
Probability of outputting 0

- Starting state with `S[2]=0` happens with probability $1/256$. This outputs `0` unless `b = 2`, thus with probability $254/255$.
- No other strong biases – so for any other starting state the probability to output some value `v` is $1/256$, for a total of $255/(256)^2$ (plus a tiny bit for `S[2]=0, S[1]=2`).
- `0` gets output with probability $255/(256)^2 + (1/256)(254/255)$. This is about twice as high, matching the experiment.

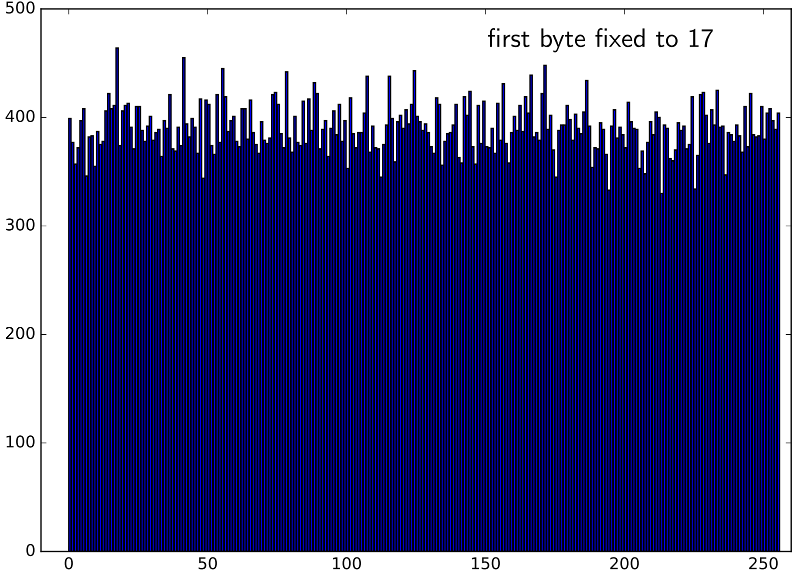
Plotting the first output byte, 100 000 runs



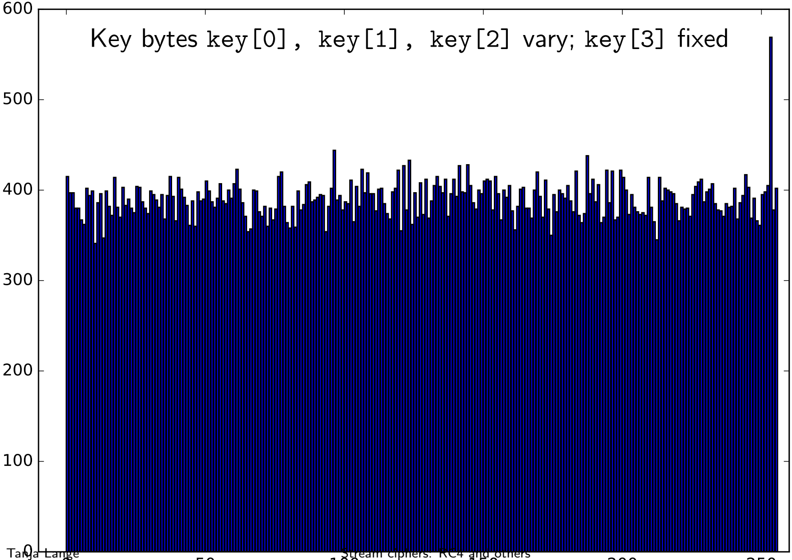
Plotting the first output byte, 100 000 runs



Plotting the first output byte, 100 000 runs



Plotting third output byte + `key[0]` + `key[1]` + `key[2]` + `key[3]`, 100 000 runs



Biases – and what they mean in practice

- Second output byte is more likely to be 0:
  guess that second byte in ciphertext matches plaintext byte.
- First output byte is biased towards the first key byte:
  If plaintext has fixed formatting / known start, learn first key byte.
- Sum of third output byte and `key[0] + key[1] + key[2] + key[3]` is biased towards 253:

Biases – and what they mean in practice

- Second output byte is more likely to be 0:
  guess that second byte in ciphertext matches plaintext byte.
- First output byte is biased towards the first key byte:
  If plaintext has fixed formatting / known start, learn first key byte.
- Sum of third output byte and `key[0] + key[1] + key[2] + key[3]` is biased towards 253:
  Note that RC4 has no place for the IV, so WEP redefines the key to be 3 bytes of IV, followed by the actual key.
  This means that for WEP the first 3 bytes vary and are known,
  the 4th is fixed and interesting.

Biases – and what they mean in practice

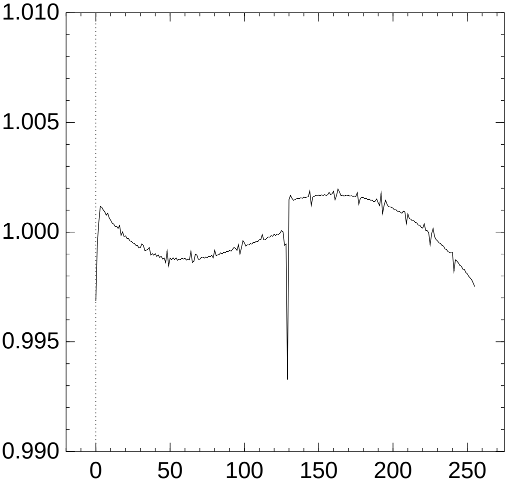
- Second output byte is more likely to be 0:
  guess that second byte in ciphertext matches plaintext byte.
- First output byte is biased towards the first key byte:
  If plaintext has fixed formatting / known start, learn first key byte.
- Sum of third output byte and `key[0] + key[1] + key[2] + key[3]` is biased towards 253:
  Note that RC4 has no place for the IV, so WEP redefines the key to be 3 bytes of IV, followed by the actual key.
  This means that for WEP the first 3 bytes vary and are known,
  the 4th is fixed and interesting.
- Similar relations due to Fluhrer, Mantin, and Shamir and to Klein recover the next key bytes.
- WEP is broken with few samples, see Aircrack-ng for fully worked out "password recovery".

Biases – and what they mean in practice

- Second output byte is more likely to be 0:
  guess that second byte in ciphertext matches plaintext byte.
- First output byte is biased towards the first key byte:
  If plaintext has fixed formatting / known start, learn first key byte.
- Sum of third output byte and `key[0] + key[1] + key[2] + key[3]` is biased towards 253:
  Note that RC4 has no place for the IV, so WEP redefines the key to be 3 bytes of IV, followed by the actual key.
  This means that for WEP the first 3 bytes vary and are known,
  the 4th is fixed and interesting.
- Similar relations due to Fluhrer, Mantin, and Shamir and to Klein recover the next key bytes.
- WEP is broken with few samples, see Aircrack-ng for fully worked out “password recovery”.

More biases, z_i is ith output byte

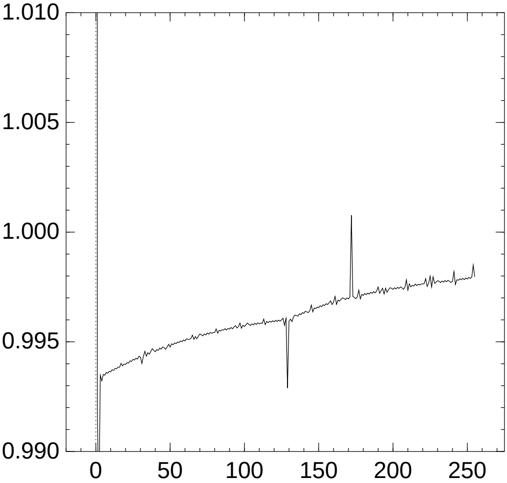
Graph of $256\,\Pr[z_1 = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

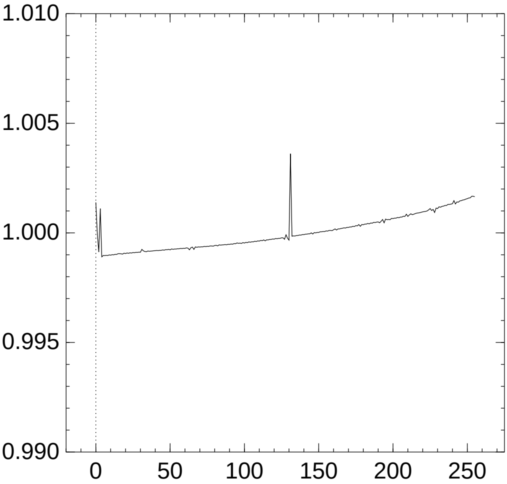
Graph of $256\,\mathrm{Pr}[z_2 = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

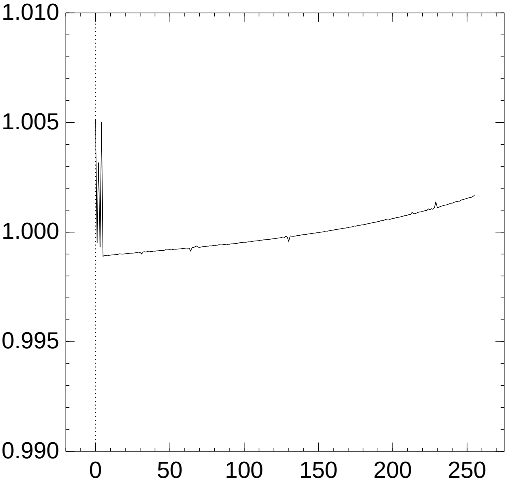
Graph of $256\,\mathrm{Pr}[z_3 = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

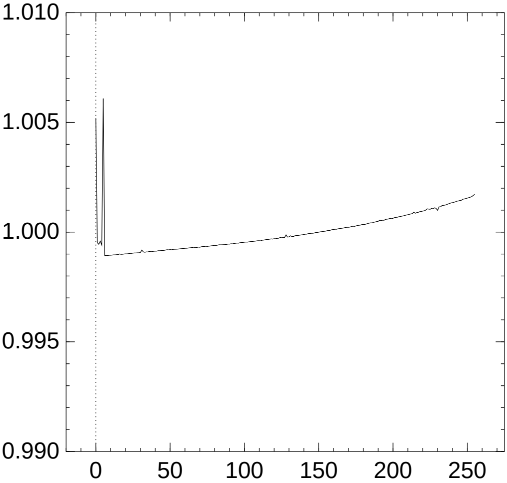
Graph of $256\,\Pr[z_4 = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

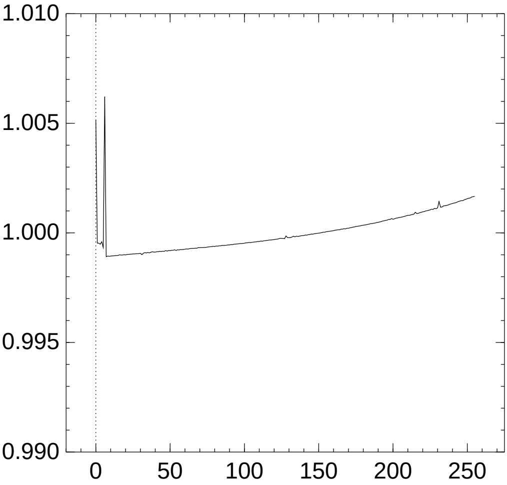
Graph of $256\,\Pr[z_5 = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

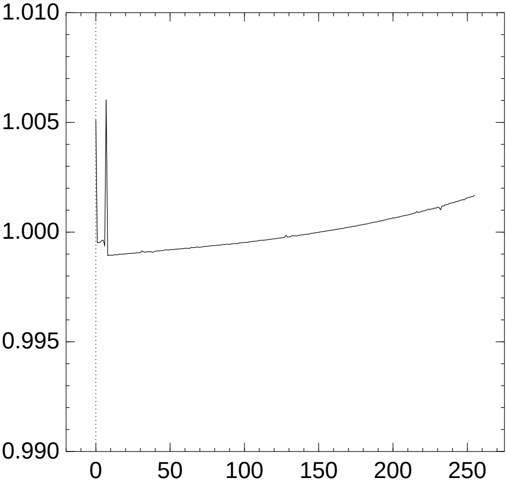
Graph of $256\,\Pr[z_6 = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

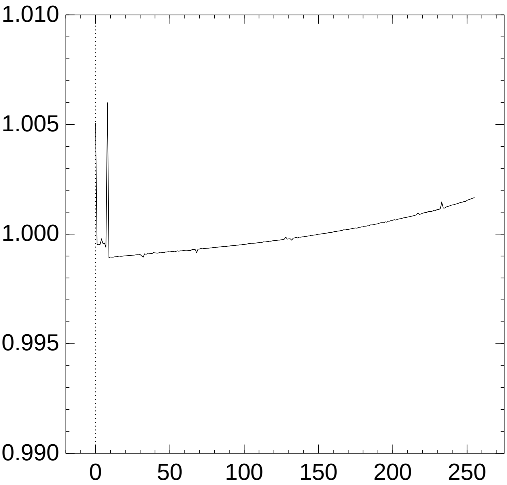
Graph of $256\,\Pr[z_7 = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

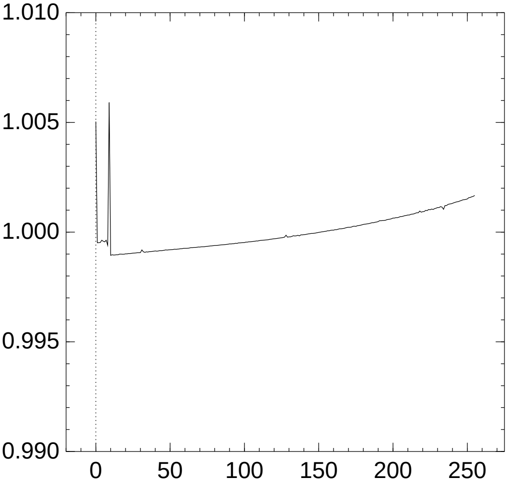
Graph of $256\,\Pr[z_8 = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is *i*th output byte

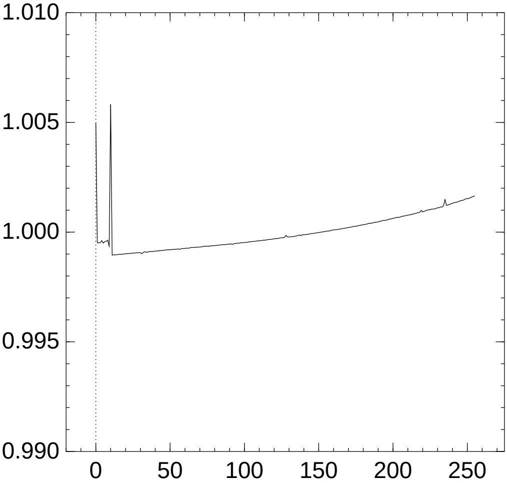
Graph of $256\,\Pr[z_9 = x]$:



From `https://cr.yp.to/talks/2013.03.12/slides.pdf`

More biases, z_i is ith output byte

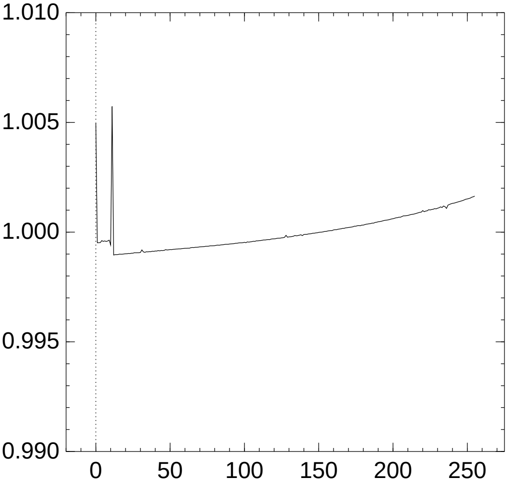
Graph of $256\,\Pr[z_{10} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

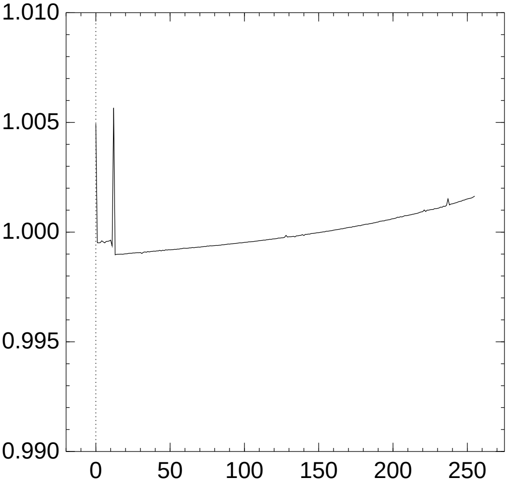
Graph of $256\,\Pr[z_{11} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

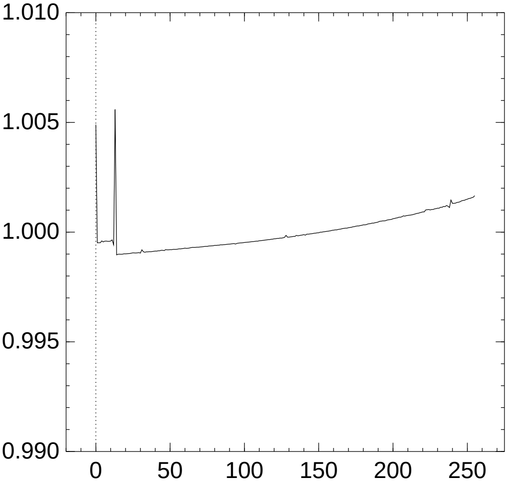
Graph of $256\,\Pr[z_{12} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

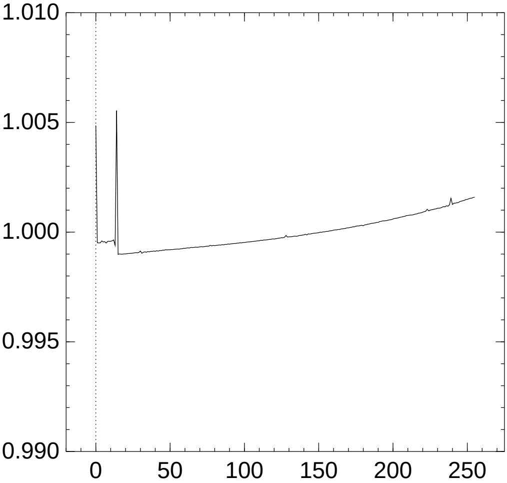
Graph of $256\,\Pr[z_{13} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is *i*th output byte

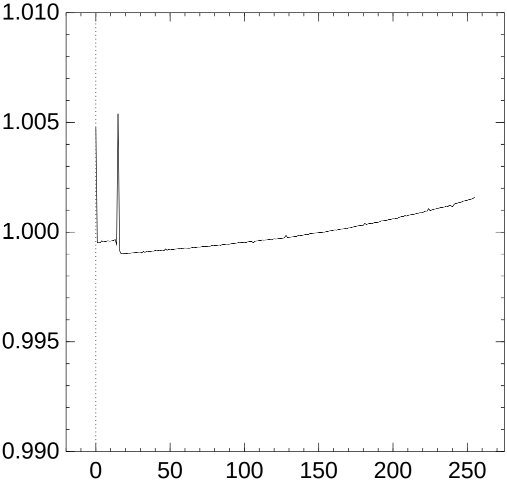
Graph of $256\,\Pr[z_{14} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

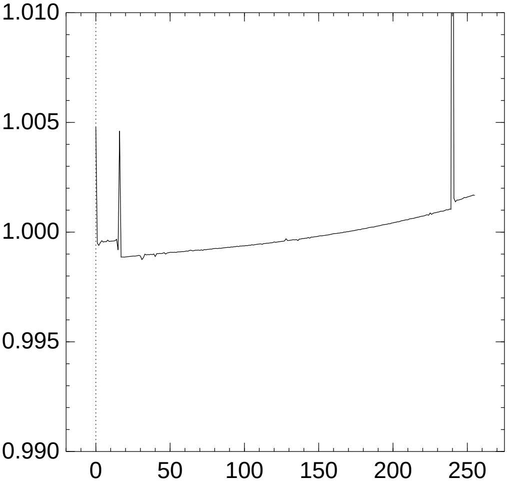
Graph of $256\,\mathrm{Pr}[z_{15} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

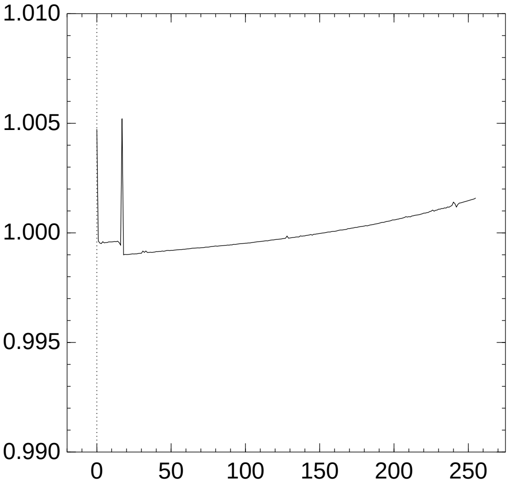
Graph of $256\,\Pr[z_{16} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

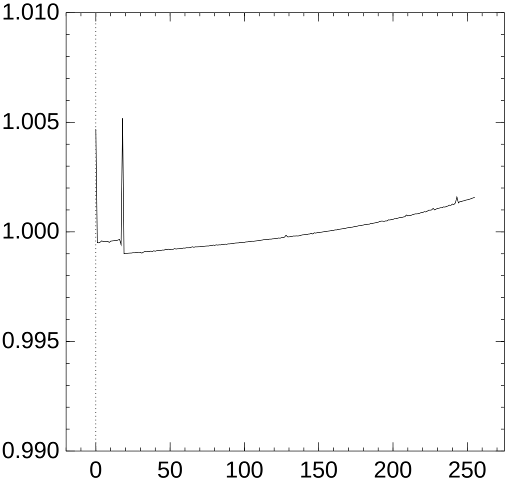
Graph of $256\,\Pr[z_{17} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

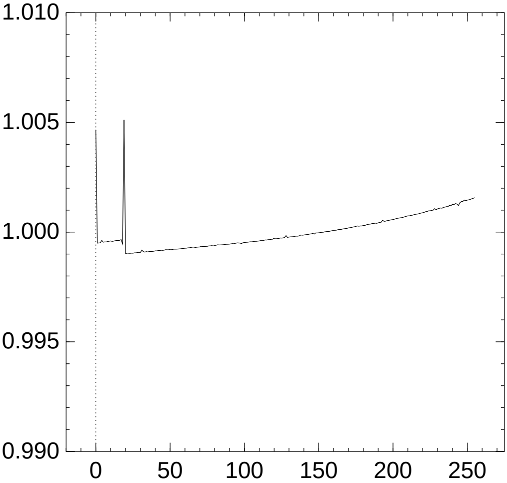
Graph of $256\,\Pr[z_{18} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

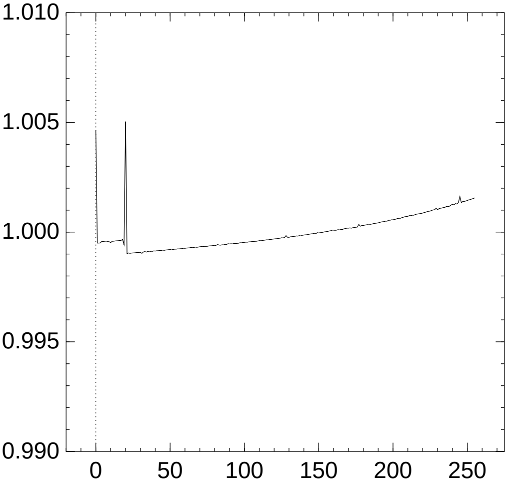
Graph of $256\,\Pr[z_{19} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

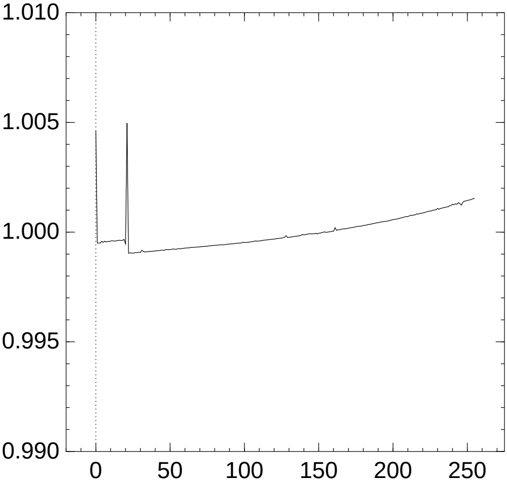
Graph of $256\,\Pr[z_{20} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

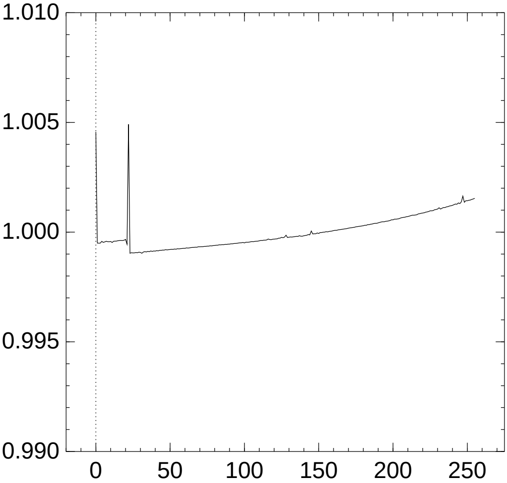
Graph of $256\,\Pr[z_{21} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

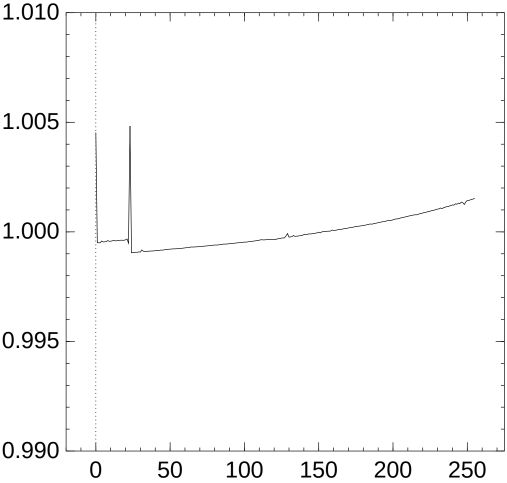
Graph of $256\,\Pr[z_{22} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

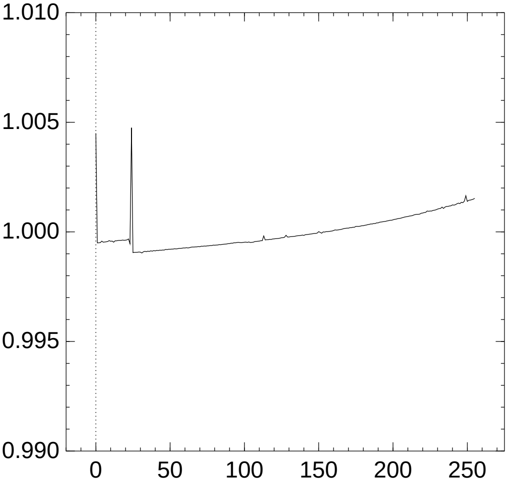
Graph of $256\,\mathrm{Pr}[z_{23} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

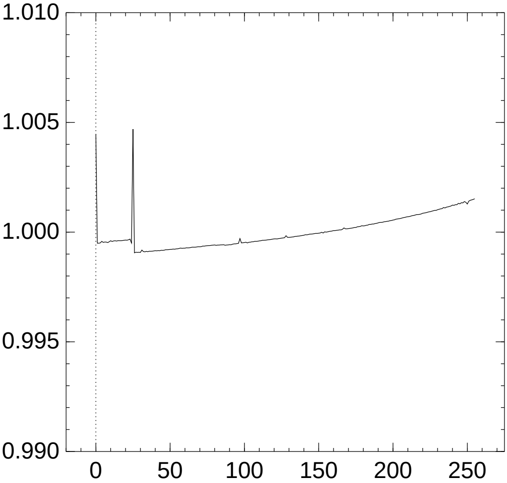
Graph of $256\,\Pr[z_{24} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

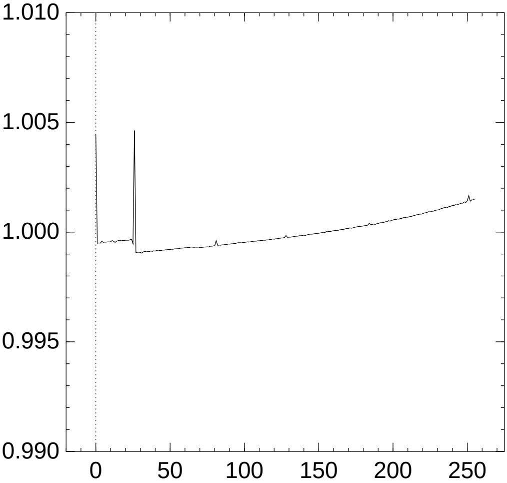
Graph of $256\,\Pr[z_{25} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

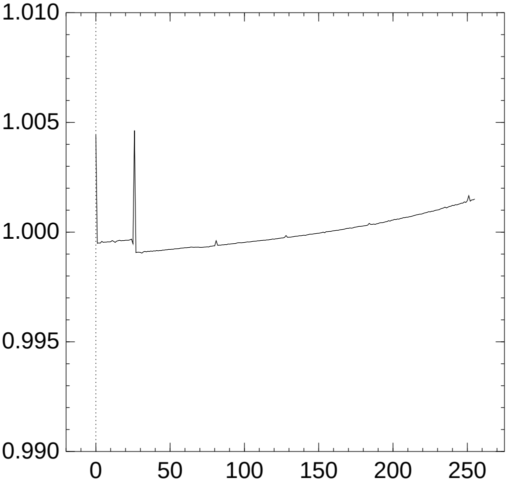
Graph of $256\,\Pr[z_{26} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

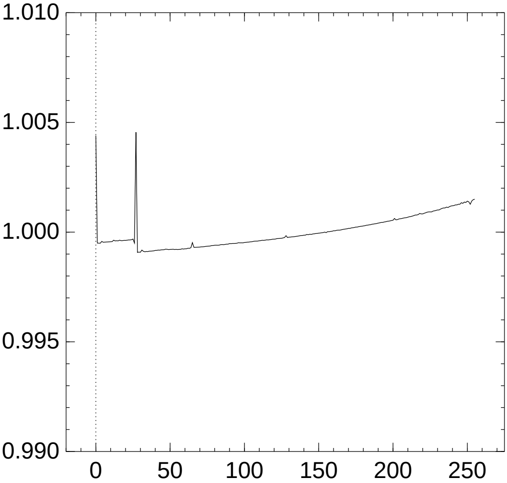
Graph of $256\,\Pr[z_{26} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

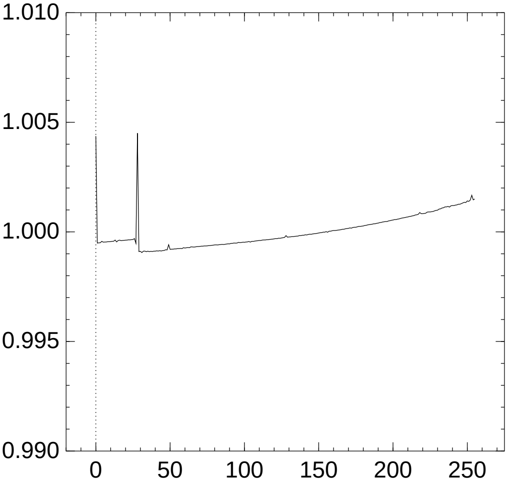
Graph of $256\,\Pr[z_{27} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

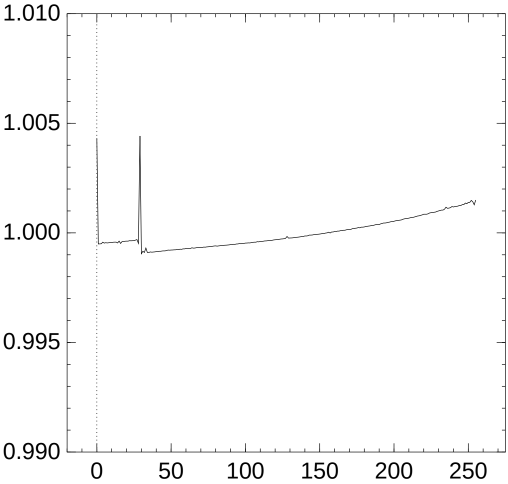
Graph of $256\,\Pr[z_{28} = x]$:



From `https://cr.yp.to/talks/2013.03.12/slides.pdf`

More biases, z_i is ith output byte

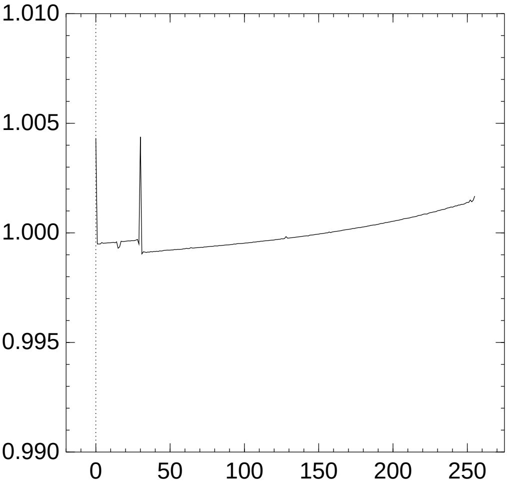
Graph of $256\,\Pr[z_{29} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

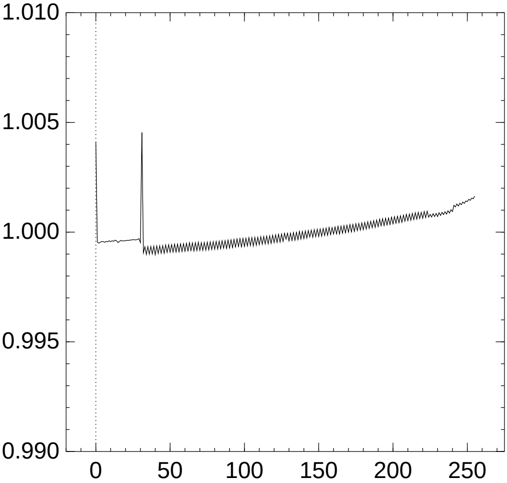
Graph of $256\,\mathrm{Pr}[z_{30} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

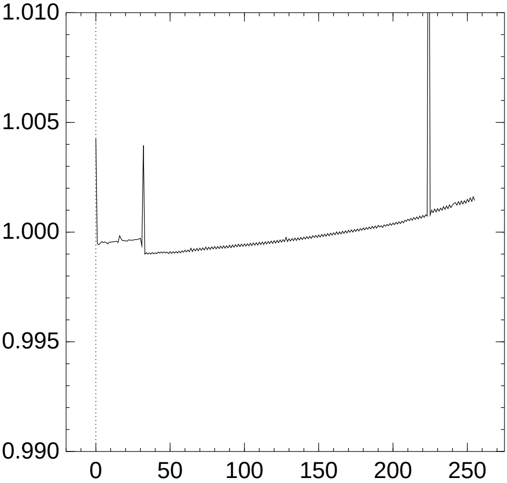
Graph of $256\,\Pr[z_{31} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

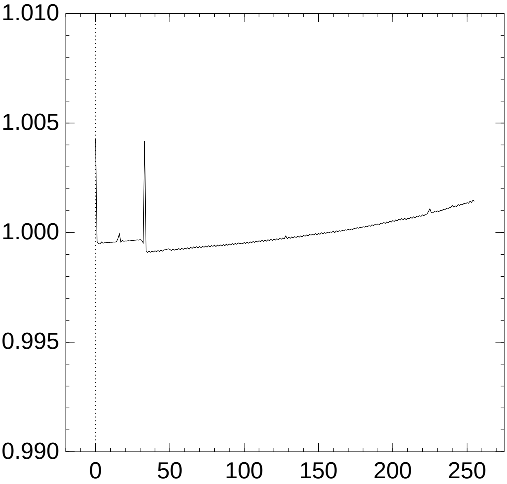
Graph of $256\,\Pr[z_{32} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

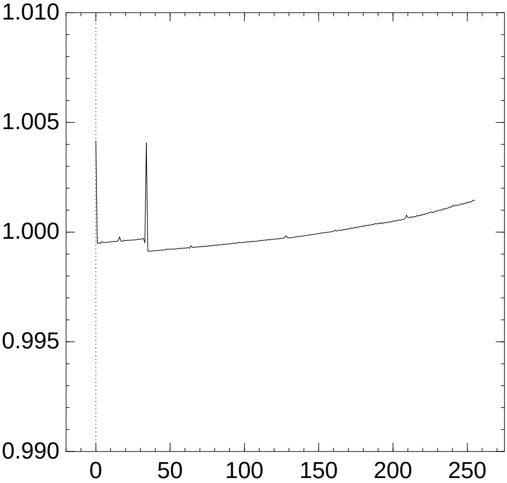
Graph of $256\,\mathrm{Pr}[z_{33} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

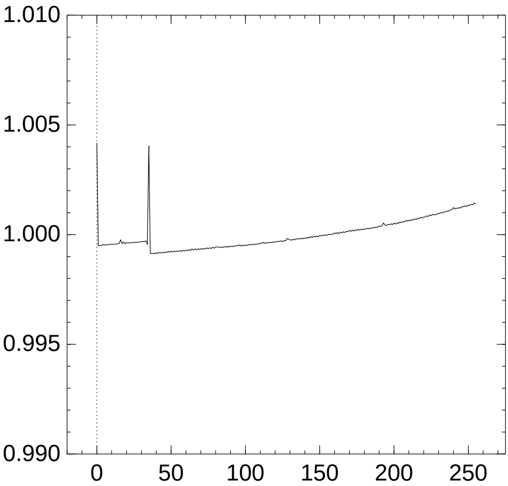
Graph of $256\,\Pr[z_{34} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

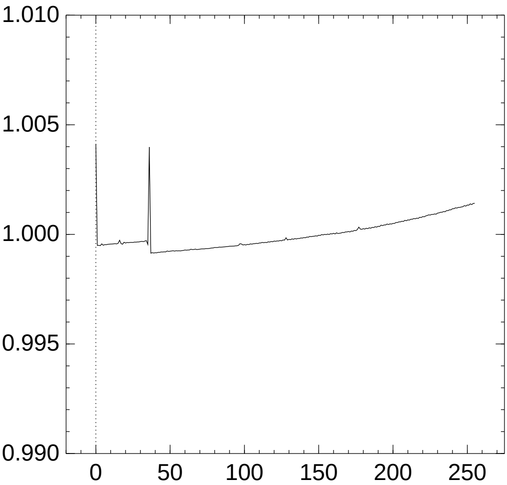
Graph of $256\,\Pr[z_{35} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

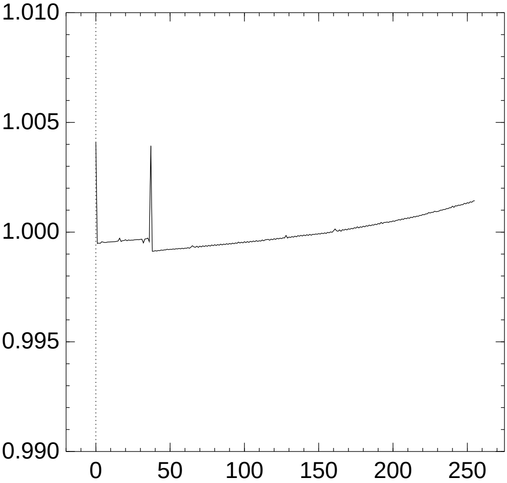
Graph of $256\,\Pr[z_{36} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

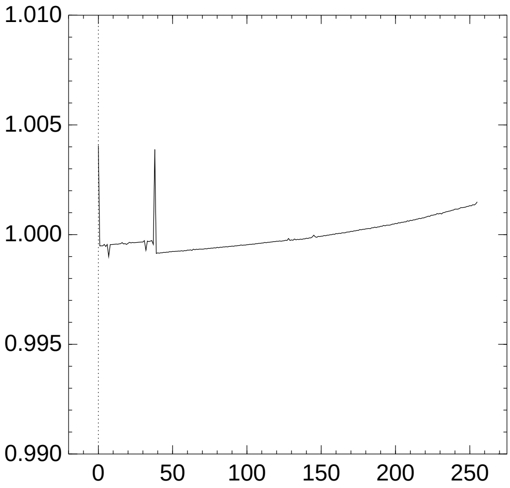
Graph of $256\,\mathrm{Pr}[z_{37} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

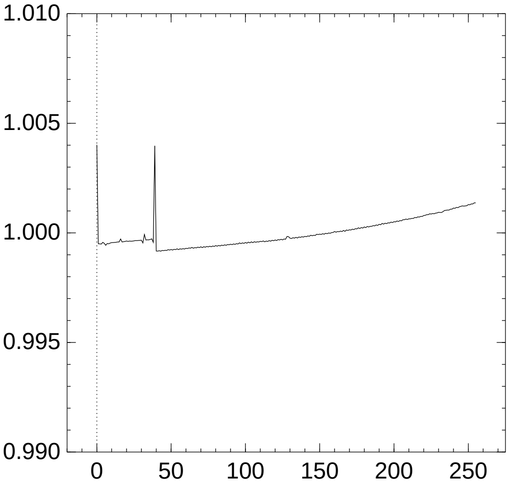
Graph of $256\,\Pr[z_{38} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

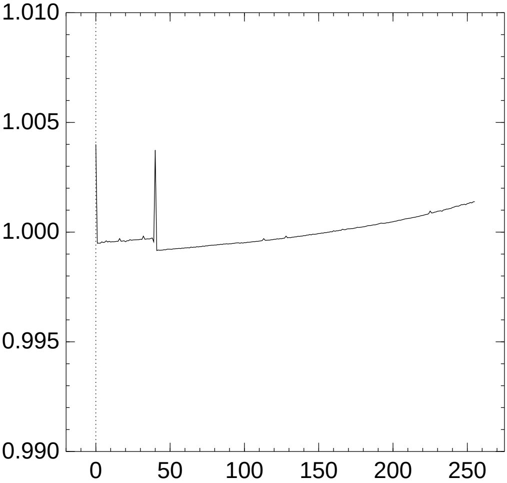
Graph of $256\,\Pr[z_{39} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

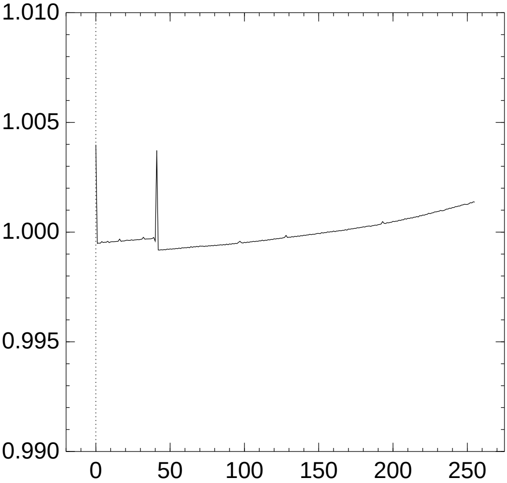
Graph of $256\,\Pr[z_{40} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

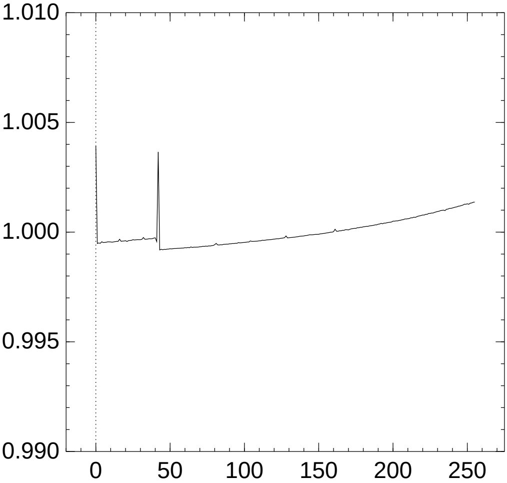
Graph of $256\,\Pr[z_{41} = x]$:



From `https://cr.yp.to/talks/2013.03.12/slides.pdf`

More biases, z_i is ith output byte

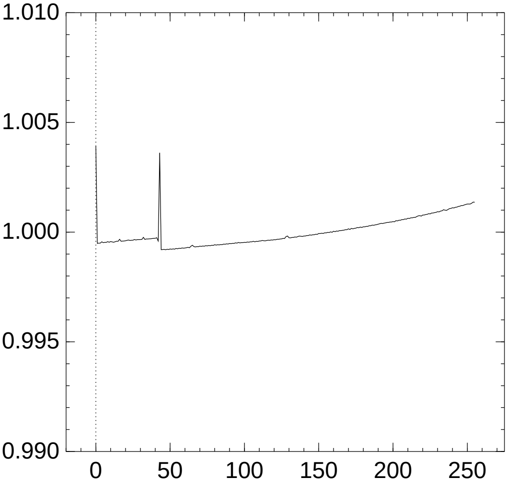
Graph of $256\,\Pr[z_{42} = x]$:



From `https://cr.yp.to/talks/2013.03.12/slides.pdf`

More biases, z_i is ith output byte

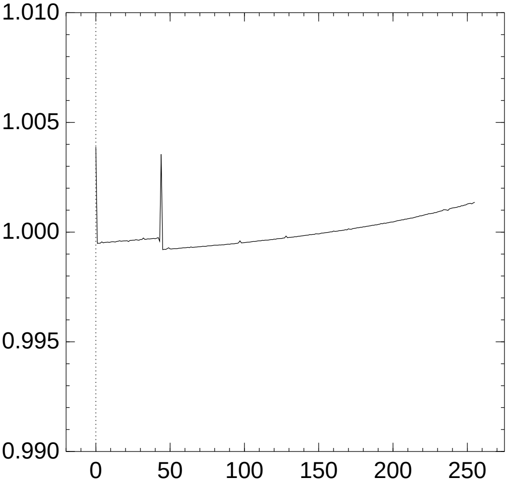
Graph of $256\,\Pr[z_{43} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

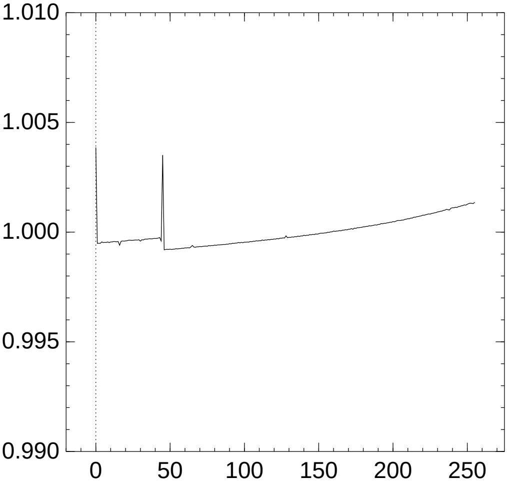
Graph of $256\,\Pr[z_{44} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

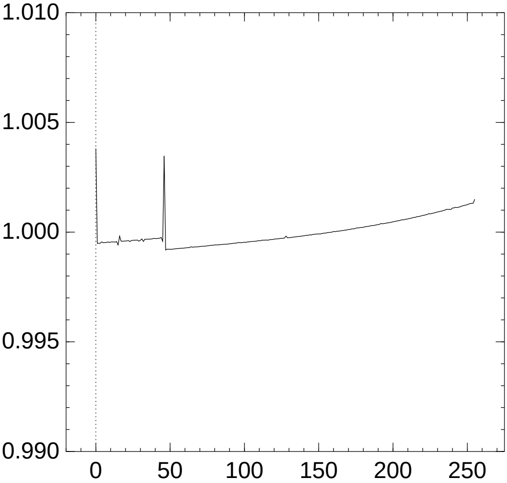
Graph of $256\,\Pr[z_{45} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

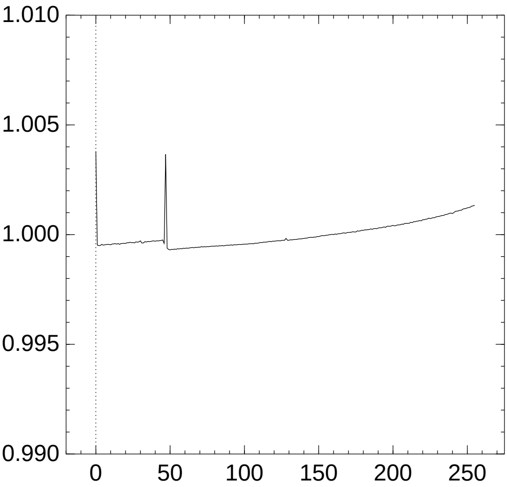
Graph of $256\,\Pr[z_{46} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

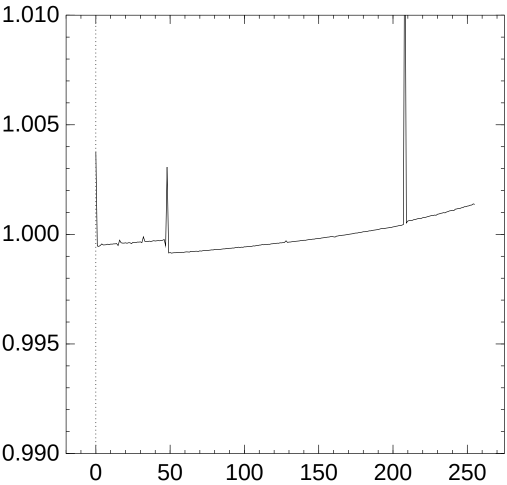
Graph of $256\,\Pr[z_{47} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

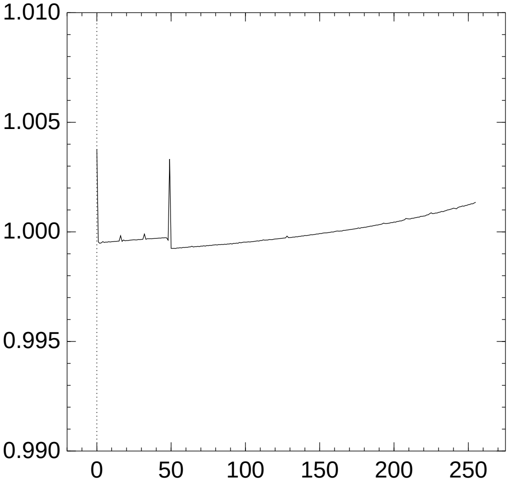
Graph of $256\,\mathrm{Pr}[z_{48} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

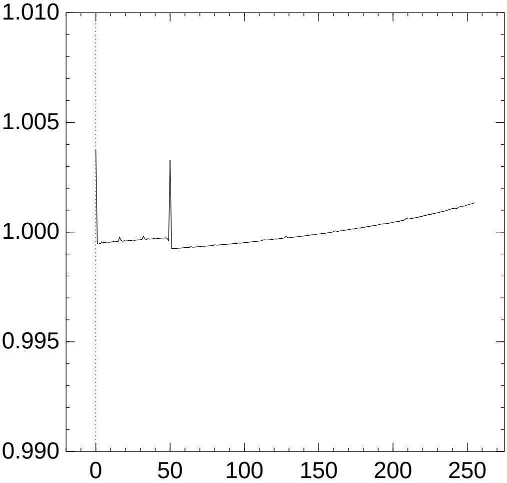
Graph of $256\,\Pr[z_{49} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

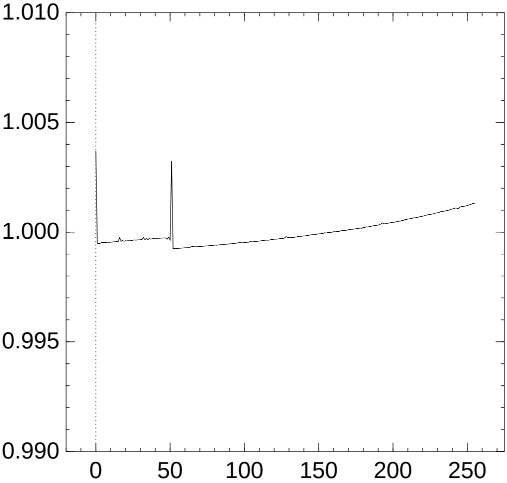
Graph of $256\,\Pr[z_{50} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

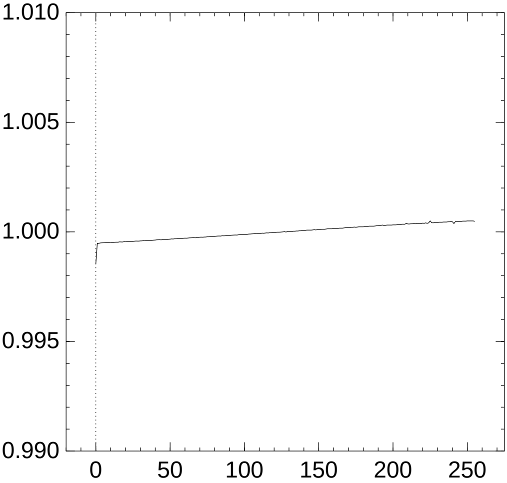
Graph of $256\,\Pr[z_{51} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

More biases, z_i is ith output byte

Graph of $256\,\Pr[z_{256} = x]$:



From https://cr.yp.to/talks/2013.03.12/slides.pdf

Stream ciphers

- In 2013 RC4 was the preferred symmetric encryption in TLS 1.0. (Seen as better of two evils.)
- Rivest recommends to discard some output bytes (enough to avoid biases?)
- 2013 AlFardan, Bernstein, Paterson, Poettering, Schuldt, “On the security of RC4 in TLS” showed 2^{32} plaintext recovery. Many plaintext bytes recovered with 2^{24} ciphertexts.

Stream ciphers

- In 2013 RC4 was the preferred symmetric encryption in TLS 1.0. (Seen as better of two evils.)
- Rivest recommends to discard some output bytes (enough to avoid biases?)
- 2013 AlFardan, Bernstein, Paterson, Poettering, Schuldt, "On the security of RC4 in TLS" showed 2^{32} plaintext recovery. Many plaintext bytes recovered with 2^{24} ciphertexts.
- Many better candidates produced in eSTREAM competition. e.g., ChaCha20 (used in TLS 1.2 and 1.3).

Stream ciphers

- In 2013 RC4 was the preferred symmetric encryption in TLS 1.0.
  (Seen as better of two evils.)
- Rivest recommends to discard some output bytes
  (enough to avoid biases?)
- 2013 AlFardan, Bernstein, Paterson, Poettering, Schuldt, “On the security of RC4 in TLS” showed 2^{32} plaintext recovery.
  Many plaintext bytes recovered with 2^{24} ciphertexts.
- Many better candidates produced in eSTREAM competition.
  e.g., ChaCha20 (used in TLS 1.2 and 1.3).
- Warning: Stream ciphers protect only confidentiality.
  They do not achieve integrity and authenticity.
  Flipping bit i in the ciphertext flips bit i in the plaintext.