



# RC4-Algorithm

Quentin Stickler, B.Sc.

24. Juli 2024

# Agenda

- 1 General info
- 2 RC4 Algorithm in detail
- 3 Attacking RC4
- 4 Preventing attacks

# General info

## History

- Stream cipher with variable key-size length
- Used to be the most widely used stream cipher in software applications
- Invented in 1987 by Ron Rivest
- Kept secret but got leaked in 1994
- Easy to implement and quite fast (Encryption up to 10x faster than DES)
- ...but also very vulnerable
- Better alternatives have been invented
- Now only used in private projects due to its simplicity and performance

# RC4 Algorithm

## How does it work?

- Consists of two parts
- Part 1: Key Scheduling Algorithm (**KSA**)
- Part 2: Pseudo Random Number Generator Algorithm (**PRGA**)
- Used in the algorithm:
- S-Box (Array) with length of 256
- K-Box with repeating key entries
- Two 8-byte sized counters  $i$  and  $j$

# RC4 Initialization

## Part One: Filling S-Box and T-Box

- Counters  $i$  and  $j$  set to 0
- Linear filling of the S-Box from 0 to 255 ( $S[0] = 0, S[1] = 1 \dots$ )
- Following loop will be run:
- State space thus:  $(2^8)^2 * 256! \approx 2^{1700}$  (Question 6)

```
1  for x in range(256):  
2      sbox[x] = x  
3      kbox[x] = key[x % len(key)]  
4
```

# Initialization

## Example

- Text = "TestText"
- Key = "TestKey"
- S-Box = [0, 1, 2, 3 ..., 255]
- Initialization of T-Box:
  - ▶ Keylength = 7
  - ▶ Ascii-Text = 84 101 115 116 75 101 121

84	101	115	116	75	101	121
84	101	115	116	75	101	121
...	...	...	...	...	...	...
...	...	...	84	101	115	116

# RC4 Initialization

## Part Two: Permutation

- Permutate S-Box based on given key
- We always use modulo  $n = 256$  because of the given length

```
1     j = 0
2     for i in range(256):
3         j = (j + sbox[i] + kbox[i]) % 256
4         Swap(sbox[i], sbox[j])
5     return sbox
6
```

- At the end: (Pseudo-)randomly generated S-Box

# Permutation Example

- S-Box Initialization:

0	1	2	3	4	5	6
...	...	...	...	...	...	...
249	250	251	252	253	254	255

- $i = 0$
- $j = (j + S[i] + T[i]) \bmod(256)$
- $j = (84 + 0 + 84) \bmod(256) = 168 \bmod(256) = 168$
- Swap  $S[i](0)$  and  $S[j](84)$
- $S[i] = 84, S[j] = 0$



# Permutation Example Cont'd

84	1	2	3	4	5	6
...	...	...	...	...	...	...
80	81	82	83	0	85	86
...	...	...	...	...	...	...
249	250	251	252	253	254	255

- $i = 1$
- $j = (j + S[i] + T[j]) \bmod(256)$
- $j = (186 + 1 + 101) \bmod(256) = 288 \bmod(256) = 32$
- Swap  $S[i](1)$  and  $S[j](186)$
- $S[i] = 186, S[j] = 1$

# Permutation Example Cont'd

84	186	2	3	4	5	6
...	...	...	...	...	...	...
80	81	82	83	0	85	86
...	...	...	...	...	...	...
249	250	251	252	253	254	255

- $i = 2$
- $j = (j + S[i]) + T[i] \bmod(256)$
- $j = (47 + 2 + 115) \bmod(256) = 126 \bmod(256) = 126$
- Swap  $S[i]$  (1) and  $S[j]$  (47)
- $S[i] = 47, S[j] = 2$

# Permutation Example

## Final S-Box Form

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

- Result = Permuted S-Box
- All numbers from 0 – 255 in "random" places

# Keystream Generator

- Generate keystream depending on length of given plaintext

```
1     keystream = []
2     i = 0
3     j = 0
4     for x in range(len(text)):
5         i = (1 + i) % 256
6         j = (sbox[i] + j) % 256
7
8         Swap(sbox[i], sbox[j])
9         keystream.append(sbox[(sbox[i] + sbox[j]) % 256])
10    return keystream
11
```

# Keystream Generator

Example,  $i = 0$

- $i = 0, j = 0$
- $i = (0 + 1) \bmod 256 = 1$
- $j = (0 + 186) \bmod 256 = 186 \bmod 256 = 186$
- Swap  $S[i]$  (186) and  $S[j]$  (202)
- $t = (202 + 186) \bmod 256 = 388 \bmod 256 = 132$
- keystream = [132, ]

# Keystream Generator

Example,  $i = 1$

- $i = 1, j = 186$
- $i = (1 + 1) \bmod 256 = 2$
- $j = (186 + 47) \bmod 256 = 233 \bmod 256 = 233$
- Swap  $S[i]$  (47) and  $S[j]$  (11)
- $t = (47 + 11) \bmod 256 = 58 \bmod 256 = 58$
- keystream = [132, 58, ]

# Keystream Generator

Example,  $i = 2$

- $i = 2, j = 233$
- $i = (2 + 1) \bmod 256 = 3$
- $j = (233 + 208) \bmod 256 = 451 \bmod 256 = 185$
- Swap  $S[i]$  (208) and  $S[j]$  (90)
- $t = (208 + 90) \bmod 256 = 298 \bmod 256 = 42$
- keystream = [132, 58, 42, ....]
- Final keystream = [132, 58, 42, 7, 129, 233, 245, 149]

# Encryption

- Plaintext XOR keystream
- Plaintext = "TestText" = [84, 101, 115, 116, 84, 101, 120, 116]
- Binary: 01010100 01100101 01110011 01110100 01010100 01100101 01111000  
01110100
- Keystream = [132, 58, 42, 7, 129, 233, 245, 149]
- = 10000100 0111010 00101010 0000111 10000001 11101001 11110101 10010101
- 01010100 01100101 01110011 01110100 01010100 01100101 01111000 01110100  
XOR
- 10000100 00111010 00101010 00000111 10000001 11101001 11110101 10010101  
=
- 11010000 01011111 01011001 01110011 11010101 10001100 10001101 11100001



# Decryption

- Ciphertext XOR keystream
- Plaintext = "TestText"
- Binary: 01010100 01100101 01110011 01110100 01010100 01100101 01111000  
01110100
- Keystream = [132, 58, 42, 7, 129, 233, 245, 149] =
- 10000100 0111010 00101010 0000111 10000001 11101001 11110101 10010101
- 01010100 01100101 01110011 01110100 01010100 01100101 01111000 01110100  
XOR
- 10000100 00111010 00101010 00000111 10000001 11101001 11110101 10010101  
=
- 11010000 01011111 01011001 01110011 11010101 10001100 10001101 11100001

# WEP

## Short summary

- Wired Equivalent Protocol
- Used in IEEE 802.11 for protecting LAN users against casual eavesdropping
- Encrypt wirelessly transmitted packets
- Key used for encryption consists of a long-term key (root key) and an initialization vector
- $RC4Key = IV || rk$
- Different public IV per packet, 24-bit-sized;  $IV = (X, Y, Z)$
- 40-bit-sized secret  $rk$

# Security problems in WEP

- CRC
- RC4 IV
- IP stuff

# Attacking RC4 in WEP

- Small key sizes (40-bit *rk* and 24-bit *IV*) [?]
- IV is sent clearly together with packets
- Make use of 'weak IVs' to recover first byte of every message

# Attacks on RC4

## General process

- Cryptanalysis Trudy grabs a lot of transferred data
- Tries to catch IVs of specific forms
- Goal  $\rightarrow$  Recover the long-term key  $\rightarrow$  Then she can decrypt all the ciphertexts
- Example:  $IV = (3, N - 1, V)$ , where  $N - 1 = 255$ ,  $V$  any value  $1, \dots, 255$
- Long-term-key of the form  $(3, 255, V, K_3, K_4, K_5)$
- $K_3, K_4, K_5$  are the first unknown keybytes
- Clue is in the initialization phase

# Attacks on RC4

## Example for $K_3$

- Suppose, Trudy has recovered  $IV = (3, 255, V)$
- Used for recovering Example for  $K_3$
- Let's look at our S-Box during the initialization phase
- First,  $S$  is set to the identity permutation

$i$	0	1	2	3	4	5	...
$S_i$	0	1	2	3	4	5	...

# Attacks on RC4

## Example for $K_3$

- Now, at the first step  $i = 0$ , we compute the next  $j$
- $j = j + S_i + K_1 = 0 + 0 + 3 \bmod(256) = 3$
- Thus, the elements at position  $S_i$  and  $S_j$  are swapped

$i$	0	1	2	3	4	5	...
$S_i$	3	1	2	0	4	5	...

- At the next step  $i = 1$ , we compute  $j$  as
- $j = 3 + S_i + K_i = 3 + 1 + 255 \bmod(256) = 3$

# Attacks on RC4

## Example for $K_3$ Cont'd

$i$	0	1	2	3	4	5	...
$S_i$	3	0	2	1	4	5	...

- At the next step  $i = 2$ , we compute  $j$  as
- $j = 3 + S_2 + K_2 = 3 + 2 + V \bmod(256) = 5 + V$

$i$	0	1	2	3	4	5	...	$5 + V$	...
$S_i$	3	0	$5 + V$	1	4	5	...	2	...



# Attacks on RC4

## Example for $K_3$ : Last step

- At the next step  $i = 3$ , we compute  $j$  as
- $j = 5 + V + S_3 + K_3 = 5 + V + 1 + K_3 \bmod(256) = 6 + V + K_3$

$i$	0	1	2	3	4	5	...	$5 + V$	...	$6 + V + K_3$	...
$S_i$	3	0	$5 + V$	$6 + V + K_3$	4	5	...	2	...	1	...

- Suppose  $S_0, S_1$  and  $S_3$  will remain unchanged until step  $i = 255$
- Then, the first keystreambyte will be computed following the keystream generator algorithm

# Attacks on RC4

## Example for $K_4$ and $K_5$ : Last step

- $IV = (4, 255, V)$  for  $K_4$  after  $i = 4$  steps:

$i$	0	1	2	3	4	5	6 + V	9 + V + K3	10 + V + K3 + K4
$S_i$	4	0	6 + V	9 + V + K3	10 + V + K3 + K4	5	2	3	1

- $IV = (5, 255, V)$  for  $K_5$  after  $i = 5$  steps:

$i$	0	1	2	3	4
$S_i$	5	0	7 + V	10 + V + K3 +	14 + V + K3 + K4

5	7 + V	10 + V + K3	14 + V + K3 + K4	15 + V + K3 + K4 + K5
15 + V + K3 + K4 + K5	2	3	4	5

# RC4 Attack

## Recover $K_3$

```
1  keystream = []
2  i = 0
3  j = 0
4  for x in range(len(text)):
5      i = (i + 1) % 256
6      j = (sbox[i] + j) % 256
7      Swap(sbox[i], sbox[j])
8      keystream.append(sbox[(sbox[i] + sbox[j]) % 256])
9  return keystream
10
```

- $i = 1, j = 0$
- $K_B = (6 + V + K_3) \bmod(256)$

# RC4 Attack

## Recover $K_3$ Cont'd

- $K_B = (6 + V + K_3) \bmod(256)$
- Suppose, Trudy can guess or knows the first byte of the plaintext, she can determine  $K_3$  with:
- $\rightarrow K_3 = K_B - 6 - V \bmod(256)$

# Recovery of unknown bytes

## Theorem

Let  $K_n$  be the RC4 key value at position  $n$ . Let  $IV_n$  be a tuple of  $(n, N - 1, V)$ , where  $N = 256, V \in 0, \dots, 255, n \geq 3$  and  $k_n$  the known keystreambyte at position  $n$ . Then  $K_n = k_n - \sum_1^n x - V - (\sum_3^{n-1} K_n)$  (Question 8)

- How many IVs are sufficient to determine  $K_n$ ?
- Determine probability that  $S_0, S_1, S_3$  remain unchanged
- Probability of that:  $(\frac{253}{256})^{252} = 0.0513$
- $\rightarrow$  Approximately 5%
- What is a sufficient number of IVs in order to recover  $K_3$ ?

# RC4 Attack

## Probability of recovering $K_3$

```
1  success_probability = 0.05
2  #Win probability
3  target_probability = 0.95
4  num_trials = 1
5
6  #Go through the IVs
7  while True:
8      cumulative_probability = 1 - binom.cdf(0, num_trials, success_probability)
9      if cumulative_probability >= target_probability:
10         break
11     num_trials += 1
12     return num_trials
13
```

# RC4 Attack

## Probability of recovering $K_3$

- How many IVs needed for
- 50%  $\rightarrow$  14
- 95%  $\rightarrow$  60
- Hence, 60 often regarded as sufficient for determining  $K_3$  (Question 7)
- Hier nochmal gucken, was die Wahrscheinlichkeit ist, solche IVs zu bekommen
- Ich braeuchte laut meinem Code 5 Millionen lol

# RC4 Attack

## Probability of recovering $K_n$

- Same probability for recovering  $K_4, K_5, \dots$
- If correct  $IV$  is found

$i$	0	1	2	3	4	5	$6 + V$	$9 + V + K_3$	$10 + V + K_3 +$
$S_i$	4	0	$6 + V$	$9 + V + K_3$	$10 + V + K_3 + K_4$	5	2	3	1



# RC4 Attack

## Probability of recovering $K_n$

```
1  for x in range(plaintextlength):
2      i = (i + 1) % 256
3      j = (j + S[i]) % 256
4      currentValue = S[i]
5      S[i] = S[j]
6      S[j] = currentValue
7      t = (S[i] + S[j]) % 256
8      keystream.append(S[t])
9  return keystream
10
```

- $kB = S[t] = S[4] = 10 + V + K_3 + K_4$

# RC4 Attack

## Probability of recovering $K_n$

- $\rightarrow$  Same probability for recovering  $K_n$
- Also doable with IVs of other form
- Suppose,  $IV = (2, 253, 1)$  for recovering  $K_3$
- Then, after  $i = 3$  steps, the  $S$  –  $box$  will have the following form:

$i$	0	1	2	3	...	$3 + K_3$	...
$S_i$	0	2	1	$3 + K_3$	...	3	...

# RC4 Attack

## Determine useful IVs

### Theorem

Let  $kN$  be the keystreambyte at position  $n$  we are looking for. We denote  $IV \dagger kN$ , if the given  $IV$  is useful for the attacker to recover  $kN$ . To check if a given  $IV = (x, y, z)$  is useful for the attack, we calculate the  $s$  – box until step  $i = n$ .

$$S[i] + S[S[i]] \stackrel{?}{=} n \rightarrow IV \dagger kN.$$

- To increase our chances, we can use IVs of other forms as well
- Hier nochmal gucken, wie viele allgemein gut sind, da gabs ne gute Quelle irgendwo!!!!
- Examples for recovering  $K_3$ :  $(2, 253, 0)$  (Question 10)

# Prevention against RC4 attacks

## Many improved algorithms

- Performance ↔ Security trade-off
- **RC4+** offers best security, but 3x execution time
- Uses three layers of scrambling the s-box
- **Improved RC4**
- Focus on altering PRGA by adding XOR operations and using two S-boxes and higher speed
- **Effective RC4**
- Same KSA as Improved KSA
- IN PRGA, two output bytes are produced and XORed with plaintext bytes
- Faster and more secure
- **RC4FMS** focuses on decreasing chances of a successful FMS attack
- Adds more randomness to the first 4 bytes

# Prevention against RC4 attacks

## Many improved algorithms

- Add 256 more steps to the initialization process
- Generate 256 keystream bytes and discard them after the initialization process
- Then generate the actual keystream

