Applied Computer Sciences and Biosciences

HOCHSCHULE MITTWEIDA
University of Applied Sciences

RC4-Algorithm

Quentin Stickler, B.Sc.

24. April 2024

hs-mittweida.de

Agenda

General info

History

- Stream cipher with **variable** key-size length
- Most widely used stream cipher in software applications in the past
- 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)
- Offers a **lot of weaknesses und vulnerabilities**
- 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)**
- $S - Box$ (Array) with length of 256
- Two 8-byte sized counters i and j
- State space thus: $(2^8)^2 * 256! \approx 2^{1700}$ (Question 6)

Initialization

Part One: Filling S-Box and K-Box

- Counters i and j set to 0
- Linear filling of the S-Box from 0 to 255 ($S[0] = 0, S[1] = 1\dots$)
- Store key bytes in seperate $K - Box$

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

```

Initialization

Example

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

$$\begin{vmatrix} 84 & 101 & 115 & 116 & 75 & 101 & 121 \\ 84 & 101 & 115 & 116 & 75 & 101 & 121 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & 84 & 101 & 115 & 116 \end{vmatrix}$$

Initialization

Part Two: Permutation

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

```
j = 0
for i in range(256):
  j = (j + sbox[i] + kbox[i]) % 256
  Swap(sbox[i], sbox[j])
return sbox

```

- At the end: (Pseudo-)randomly generated S-Box Woungang and Dhurandher, 2019

Permutation Example

Keystream: [84, 101, 115, 116, 75, 101, 121**]**

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

- i = 0, j = 0
- j = (j + S[i] + K[i]) % (256)
- j = (0 + 0 + 84) % (256) = 84 % (256) = 84
- Swap S[i] (0) and S[j] (84)
- S[i] = 84, S[j] = 0

Permutation Example Continued

Keystream: [84**, 101, **115**,** 116**,** 75**,** 101**,** 121**]**

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

- i = 1, j = 84
- j = (j + S[i] + K[i]) % (256)
- j = (84 + 1 + 101) % (256) = 186 % (256) = 186
- Swap S[i] (1) and S[j] (186)
- S[i] = 186, S[j] = 1

Permutation Example Continued

Keystream: [84, 101, **115**, 116, 75, 101, 121]

$$\begin{vmatrix} 84 & 186 & 2 & 3 & 4 & 5 & 6 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 80 & 81 & 82 & 83 & 0 & 85 & 86 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 184 & 185 & 1 & 187 & 188 & 189 & 190 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 249 & 250 & 251 & 252 & 253 & 254 & 255 \end{vmatrix}$$

- i = 2, j = 186
- j = (j + S[i] + K[i]) % (256)
- j = (186 + 2 + 115) % (256) = 303 % (256) = 47
- Swap S[i] (2) 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 = Permutated S-Box
- All numbers from 0 – 255 in "random" places

Keystream Generator

Python Code

- Generate keystream depending on length of given plaintext

```
keystream = []
i = 0
j = 0
for x in range(len(text)):
  i = (1 + i) % 256
  j = (sbox[i] + j) % 256

  Swap(sbox[i], sbox[j])
  result = sbox[i] + sbox[j] % 256
  keystream.append(sbox[result])
return keystream
```

Keystream Generator

Example, i = 0

- i = 0, j = 0
- i = (0 + 1) % 256 = 1
- j = (0 + 186) % 256 = 186 % 256 = 186
- Swap S[i] (186) and S[j] (202)
- t = (202 + 186) % 256 = 388 % 256 = 132
- S[t] = 102
- keystream = [102,]

Keystream Generator

Example, i = 1

- i = 1, j = 186
- i = (1 + 1) % 256 = 2
- j = (186 + 47) % 256 = 233 % 256 = 233
- Swap S[i] (47) and S[j] (11)
- t = (47 + 11) % 256 = 58 % 256 = 58
- S[t] = 118
- keystream = [102, 118,]

Keystream Generator

Example, i = 2

- i = 2, j = 233
- i = (2 + 1) % 256 = 3
- j = (233 + 208) % 256 = 451 % 256 = 185
- Swap S[i] (208) and S[j] (90)
- t = (208 + 90) % 256 = 298 % 256 = 42
- S[t] = 53
- keystream = [102, 118, 53, ...]
- Final keystream = [102, 118, 53, 212, 66, 47, 204, 221]

Encryption

Bytewise XOR

- Plaintext = "TestText" = [84, 101, 115, 116, 84, 101, 120, 116]
- Keystream = [102, 118, 53, 212, 66, 47, 204, 221]
- Plaintext $\oplus$ Keystream =
- "0X320X130X460XA00X160X4A0XB40XA9" = [50, 19, 70,160, 22, 74, 180, 169]

Decryption

Bytewise XOR

- Ciphertext = "0X320X130X460XA00X160X4A0XB40XA9" = [50, 19, 70, 160, 22, 74, 180, 169]
- Keystream = [102, 118, 53, 212, 66, 47, 204, 221]
- Ciphertext $\oplus$ Keystream = "TestText"

Summary

RC4-Algorithm

- Split up into two parts
  - Part 1: Key Scheduling Algorithm **(KSA)**
  - Part 2: Pseudo Random Number Generator Algorithm **(PRGA)**
- $S - Box$ (Array) with length of 256
- Permutate S-box based on given key
- Create a keystream for $\oplus$ en-/decrypting texts bytewise

WEP

Short summary

- Wired Equivalent Protocol
- Used in IEEE 802.11 for protecting LAN users against eavesdropping
- Encrypt wirelessly transmitted packets
- Key used for encryption consists of a long-term key / root key (rk) 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

Outdated since 2004

- "Swiss Cheese" of protocols $\rightarrow$ lots of security vulnerabilities
- Small key sizes; only 64-bit and 128-bit encryption key sizes
- CRC-32 for detecting changes made to data
  - Useful for detecting errors but useless for validating cryptographic validation
  - Attacker can easily alter the data so that the validation check is getting verified
- Small IV sizes of 24-bit $\rightarrow 2^{24}$ possibilities ($< 17 million$)

Attacking RC4 in WEP

Utilizing IVs

- Small key sizes (40-bit *rk* and 24-bit *IV*)
- IV is sent clearly together with packets
- Make use of "weak IVs" to recover *rK* byte for byte
- **FMS attack** by Fluhrer, Shamir and Mantin in 2000

FMS attack on RC4

General process

- Cryptanalysis Trudy graps a lot of transfered data
- Goal $\rightarrow$ Recover rk $\rightarrow$ Then she can decrypt all the ciphertexts
- Tries to catch IVs of specific forms
- Example: $IV = (3, N-1, V)$, where $N-1 = 255, V$ any value $0, \ldots, 255$
- RC4-key of form $(3, 255, V, K_3, K_4, K_5, \ldots)$
- $K_3, K_4, K_5, \ldots$ are the first unknown keybytes
- Exploiting the initialization phase

Attacks on RC4

Example for K_3

- Suppose, Trudy has recoverd $IV = (3, 255, V)$
- Used for recovering K_3
- Study S-Box during the initialization phase
- First, S is set to the identitity permutation

i	0	1	2	3	4	5	$\dots$
S_i	0	1	2	3	4	5	$\dots$

Attacks on RC4

Example for K_3 with $IV = (3, 255, V)$

- At the first step $i = 0$, we compute the next j
- $j = j + S_i + K_i = 0 + 0 + 3 \ \%(256) = 3$
- 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
- $j = 3 + S_i + K_i = 3 + 1 + 255 \ \%(256) = 3$

Attacks on RC4

Example for K_3 with $IV = (3, 255, V)$ continued

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
- $j = 3 + S_2 + K_2 = 3 + 2 + V\ \%(256) = 5 + V$

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 with $IV = (3, 255, V)$: Continued

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

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

RC4 Attack

Recover K_3 with $IV = (3, 255, V)$: Last step

```
    keystream = []
    i = 0
    j = 0
    for x in range(len(text)):
      i = (i + 1) % 256
      j = (sbox[i] + j) % 256
      Swap(sbox[i], sbox[j])
      result = sbox[i] + sbox[j] % 256
      keystream.append(sbox[result])
    return keystream

```

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

RC4 Attack

Recover K_3 Continued

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

Recovery of unknwon bytes

General approach

Theorem

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

$$K_n = k_n - \sum_1^n x - V - (\sum_3^{n-1} K_n)$$

- How many IVs are sufficient to determine K_n?
- Determine probability that S_0, S_1, S_n remain unchanged

RC4 Attack

Probability of recovering K_n

Theorem

Let Kn be the unknown key byte at position n, $N = 256$ and $p = N - (n+1)$. Then the probability that the values in the given $S - box$ at position S_0, S_1 and S_n will remain unchanged for p steps, equals:

$$(\frac{253}{N})^p$$

- Probability for recovering K_3: $(\frac{253}{256})^{252} = 0.0513 \approx 5\%$ (Question 9)
- What is a sufficient number of IVs in order to recover K_3?

RC4 Attack

Probability of recovering K_3

```python
    success_probability = 0.05
    #Win probability
    target_probability = 0.95
    num_trials = 1

    #Go through the IVs
    while True:
      cumulative_probability = 1 - binom.cdf(0, num_trials, success_probability)
      if cumulative_probability >= target_probability:
        break
      num_trials += 1
    return num_trials
```

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)

RC4 Attack

Probability of recovering K_n

- Probability for recovering K_4: $(\frac{253}{256})^{251} = 0.0518$
- Probability for recovering K_5: $(\frac{253}{256})^{250} = 0.0525$
- Chance gets higher as we move through the $S - box$

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

How to determine useful IVs

Theorem

Let kN *be the keystreambyte at position* n *we are looking for. We define*

$$IV \dagger kN,$$

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

Usefule IV example

For K_3 (Question 10)

$$IV = (2, 253, 1) \dagger k3?$$

Prevention against RC4 attacks

Many improved algorithms

- Performance $\leftrightarrow$ Security trade-off
- **RC4+**: Best security, but 3x execution time
  - Uses three layers of scrambling the s-box
- **Improved RC4**: Improved security and parallel execution
  - Focus on altering PRGA by adding $\oplus$ operations and using two S-boxes
- **Effective RC4**: Faster and more secure
  - Same KSA as Improved KSA
  - IN PRGA, two output bytes are produces and XORed with plaintext bytes
- **RC4FMS**: Decreased chances of a succesful FMS attack
  - Adds more randomness to the first 4 bytes

Increase WEP security

With regrads to RC4

- Add 256 **more steps** to the initialization process and discard them afterwards
- Use alternative protocols such as **WPA2/WPA3** with other encryption algorithms
- **Increase IV sizes** to at least 32 bits $\rightarrow 2^8$ times for attacker to find collisions/useful IVs (Question 11)
- Use other hashing algorithms such as **MD5, SHA-1**

RC4 summary

Everything we have learnt

- Invented in 1987 by Ron Rivest as stream cipher with **variable key length**
- Officially **outdated** because of too many weaknesses
- Consists of two parts
  - **KSA**
  - **PRGA**
- Used in **WEP** and **SSL/TLS**, now replaced by other protocols/other encryption algorithms
- In-depth look at specific **FMS attack on RC4 in WEP**, makes use of weak IVs
- Numerous **improved RC4 variants** for better security, offer too many **trade-offs** compared to other algorithms

HOCHSCHULE MITTWEIDA
University of Applied Sciences

Thank You

Quentin Stickler, B.Sc.

qstickle@hs-mittweida.de

Hochschule Mittweida
University of Applied Sciences
Technikumplatz 17 | 09648 Mittweida
Applied Computer Sciences and Biosciences

hs-mittweida.de

Woungang, Isaac and Sanjay Kumar Dhurandher (2019). *2nd International Conference on Wireless Intelligent and Distributed Environment for Communication: WIDECOM 2019*. Vol. 27. Springer.