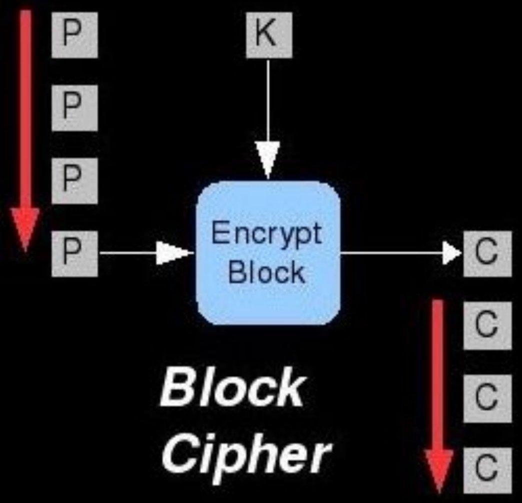
Cryptography

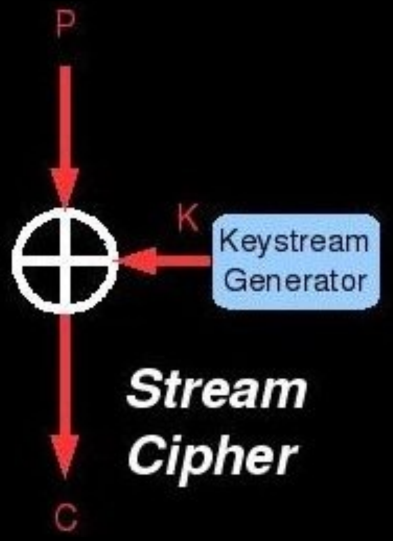
Stream Ciphers

Content

Introduction

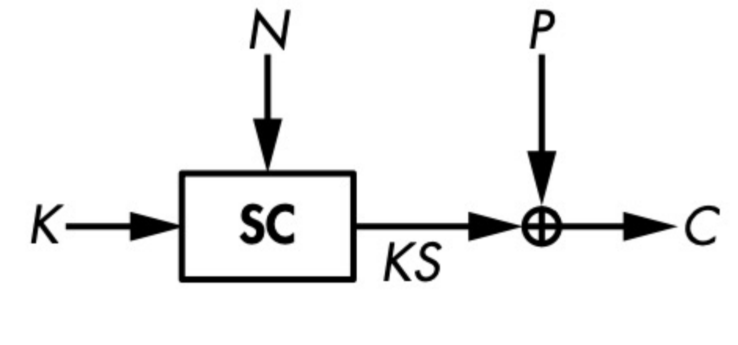
- Symmetric ciphers:
  - **Block ciphers**: mix blocks of plaintext bits to produce blocks of ciphertext bits.
  - **Stream ciphers**: generate random bits from the key and XOR it with the plaintext.
    - Similar to the OTP.
    - Akin to DRBG, not PRNG.





Introduction

- Stream ciphers take two inputs:
  - **Key**: secret – its size 128 or 256 bits.
  - **Nonce**: Not secret – its size is between 64 and 128 bits



  - Stream ciphers produce a pseudorandom stream of bits called the **keystream**.
  - **Encryption**: XOR the plaintext with the keystream.
  - **Decryption**: XOR the ciphertext with the keystream

Introduction

- Never use the same key and nonce pair twice.
  - Nonce = number used once

- If you have (K_1, N_1) encrypting a message P_1.
- You can use another pair (K_1, N_2), (K_2, N_1) or (K_2, N_2) to encrypt another message P_2.

- Using the same key and nonce will produce the same keystream.
  - If you have $C_1 = P_1 \oplus KS$ and $C_2 = P_2 \oplus KS$ and you know P_1, you can get $P_2 = C_1 \oplus C_2 \oplus P_1$

Introduction

- Types of stream ciphers
  - Stateful
  - Counter-base

Introduction
Stateful stream ciphers

- Have a secret internal state that evolves throughout keystream generation.
    1. The cipher initializes the state from the key and the nonce.
    2. Calls an $update$ function to update the state value.
    3. Produce keystream bits from the state
- Example: **RC4**

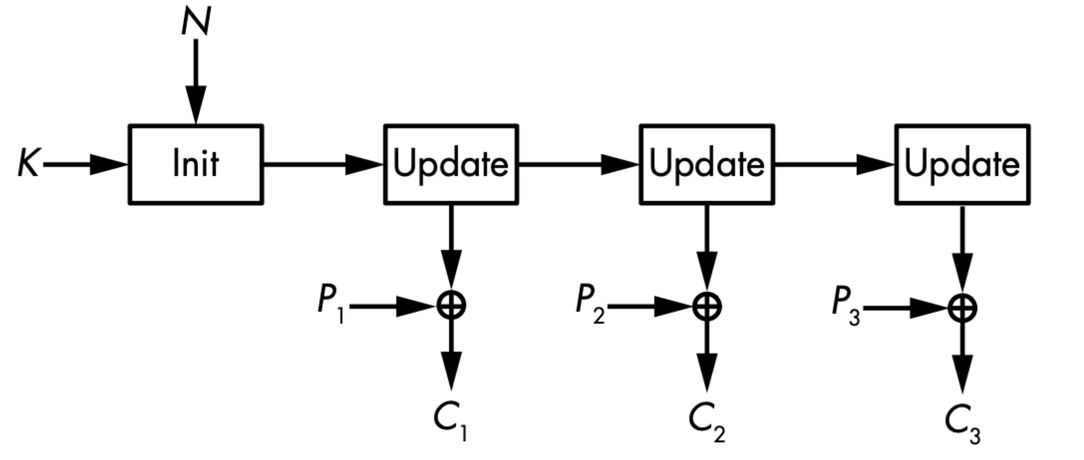


Figure 5-2: The stateful stream cipher

Introduction
Counter-based stream ciphers

- Inputs a key, a nonce, and a counter and produce chunks of keystream.

- Example: **Salsa20**

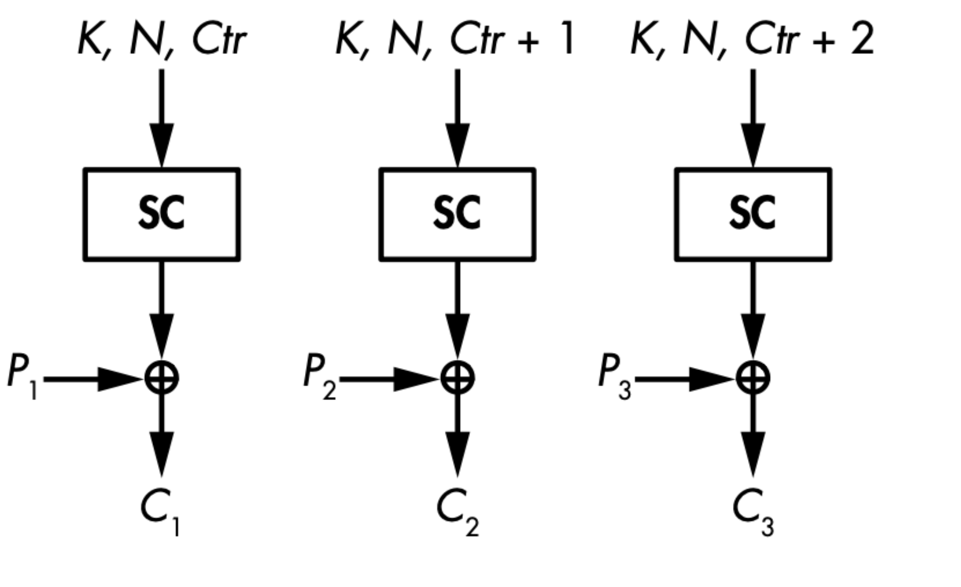
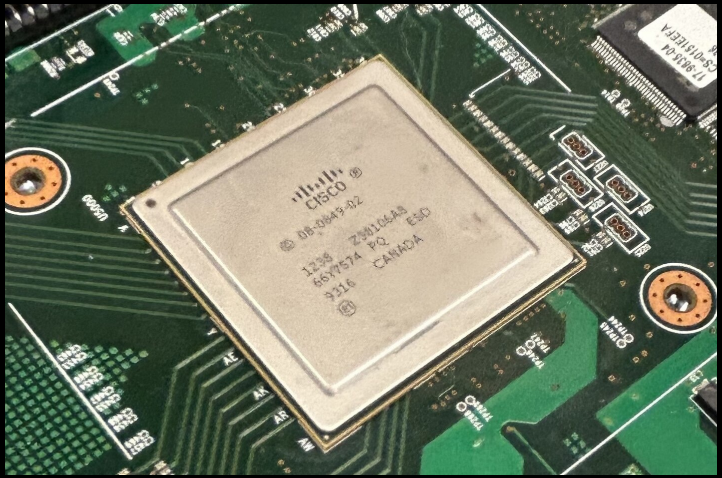
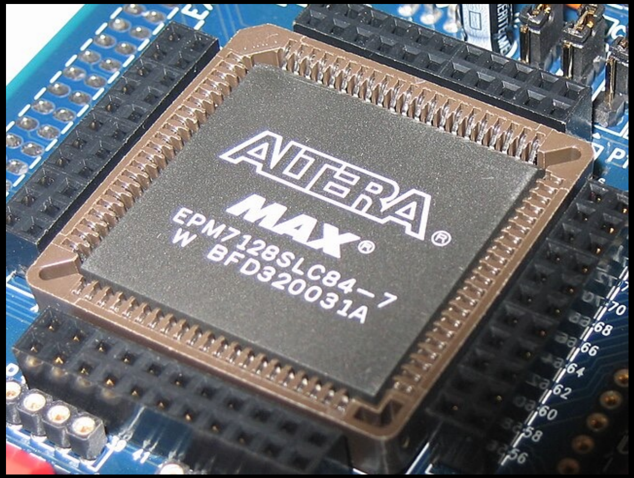


Figure 5-3: The counter-based stream cipher

Introduction

- Stream ciphers can be either:
  - Hardware-oriented stream ciphers.
    - Application Specific Integrated Circuit (ASIC)
    - Programmable Logic Device (PLD)
    - Field Programmable Gate Arrays (FPGA)
  - Software-oriented stream ciphers.





Content

Content
Introduction
Hardware-Oriented Stream Ciphers
Software-Oriented Stream Ciphers

Hardware-Oriented Stream Ciphers

- Electronic circuit that implements a cryptographic algorithm.
  - Lower cost on hardware vs block ciphers
  - Less memory, smaller chip

- Hardware stream ciphers rely on **Feedback Shift Registers (FSR)**.

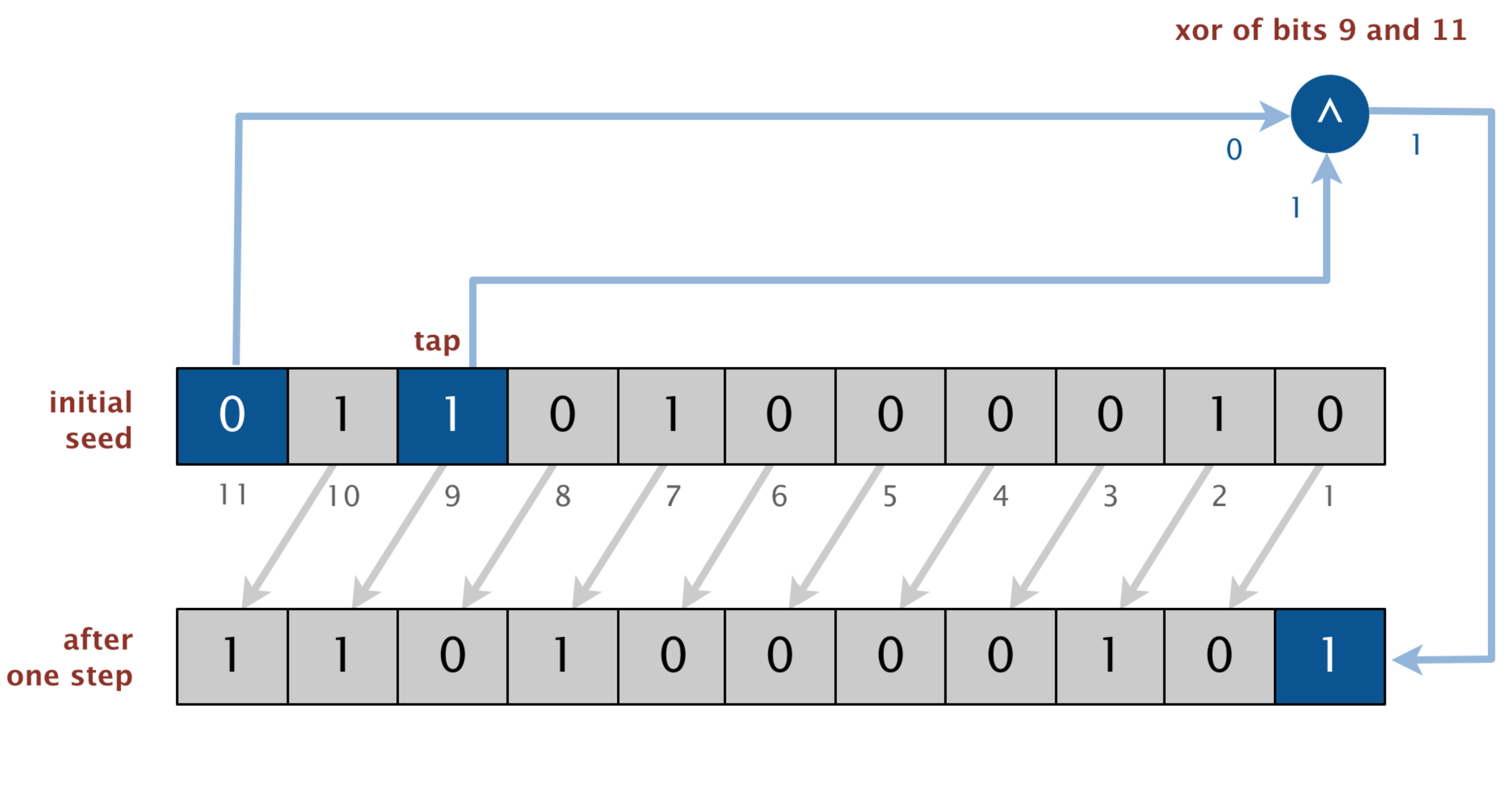
Hardware-Oriented Stream Ciphers

- Components of feedback shift register (FSR):
  - **State $\boldsymbol{R}$**: an array of bits, or registers.
  - **Feedback function $\boldsymbol{f}$**: updates the state and output bits

- FSR works as follows:
  - Shift the current state to the left by 1-bit
  - Return the shifted bit as output
  - Fill the empty position with the bit $f(R)$

$$R_{t+1} = (R_t \ll 1)\ OR\ f(R_t)$$

Hardware-Oriented Stream Ciphers



one step of an 11-bit LFSR with initial seed 01101000010

Hardware-Oriented Stream Ciphers

- Example: consider a 4-bit FSR whose feedback function f XORs all 4 bits together. The initial state is

1	1	0	0

- Shift the bits to the left and output the shifted bit

Output: 1

1	0	0	x

- Fill the rightmost position with the bit $f(1100) = 1 \oplus 1 \oplus 0 \oplus 0 = 0$

1	0	0	0

Hardware-Oriented Stream Ciphers

- The next update is as follows

1	0	0	0

Output: 11

0	0	0	x

  - Fill the rightmost position with the bit $f(1000) = 1 \oplus 0 \oplus 0 \oplus 0 = 1$

0	0	0	1

Hardware-Oriented Stream Ciphers

- The next 3 updates return three 0 bits and give the following state values:

Output				
Output: 110	0	0	1	1
Output: 1100	0	1	1	0
Output: 11000	1	1	0	0

Hardware-Oriented Stream Ciphers

- The next 3 updates return three 0 bits and give the following state values:

Output				
Output: 110	0	0	1	1
Output: 1100	0	1	1	0
Output: 11000	1	1	0	0

- We return to the initial state after 5 iterations!
- The output is the same as the initial state!

Hardware-Oriented Stream Ciphers

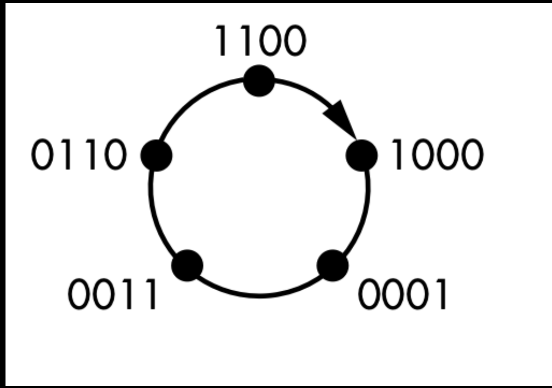
- We say that 5 is the *period* of the FSR.
  - FSR period: the number of updates needed until the FSR enters the same state again.

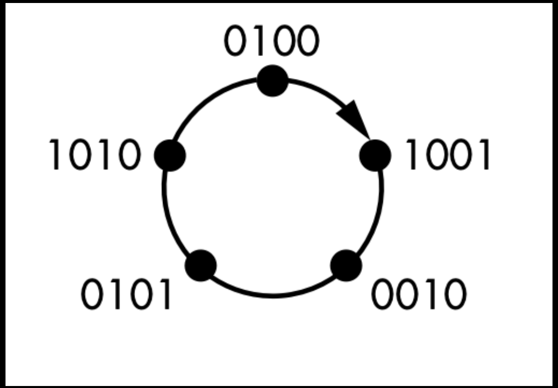
- So, if we clock the register 20 times, the output will be:

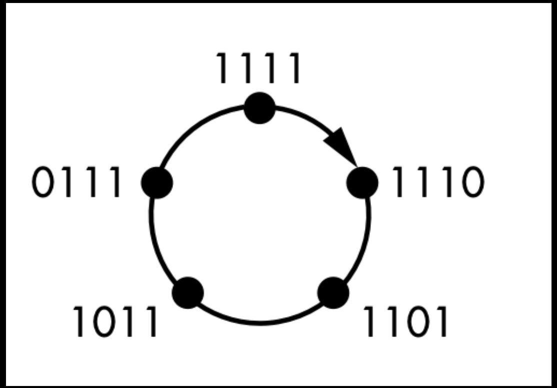
  11000110001100011000

- Patterns are bad, avoid them.
- Short periods → predictable patterns → bad FSRs

Hardware-Oriented Stream Ciphers

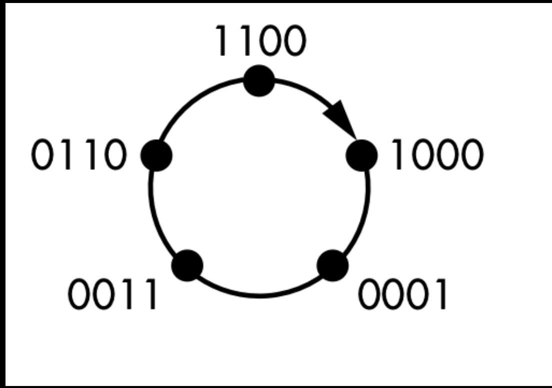


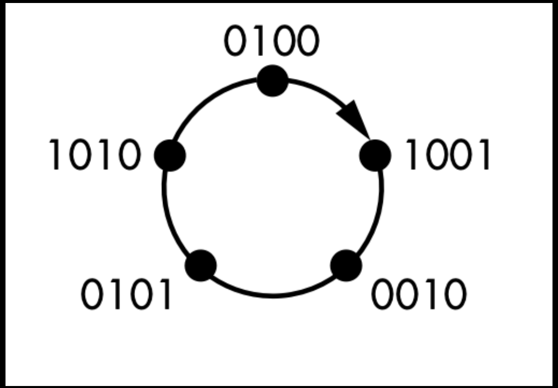


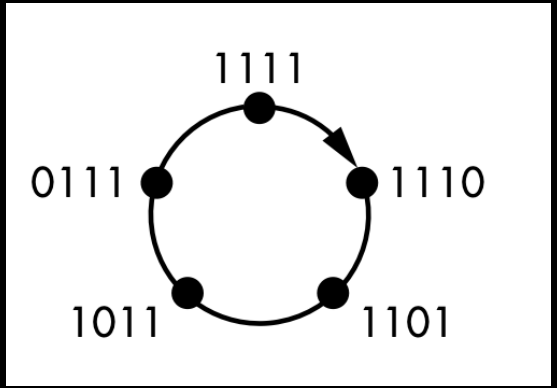


What is the period of 0000?

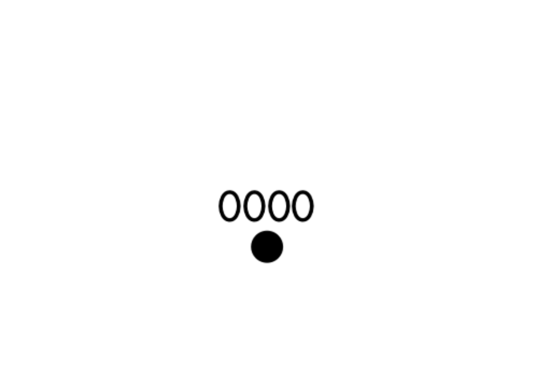
Hardware-Oriented Stream Ciphers



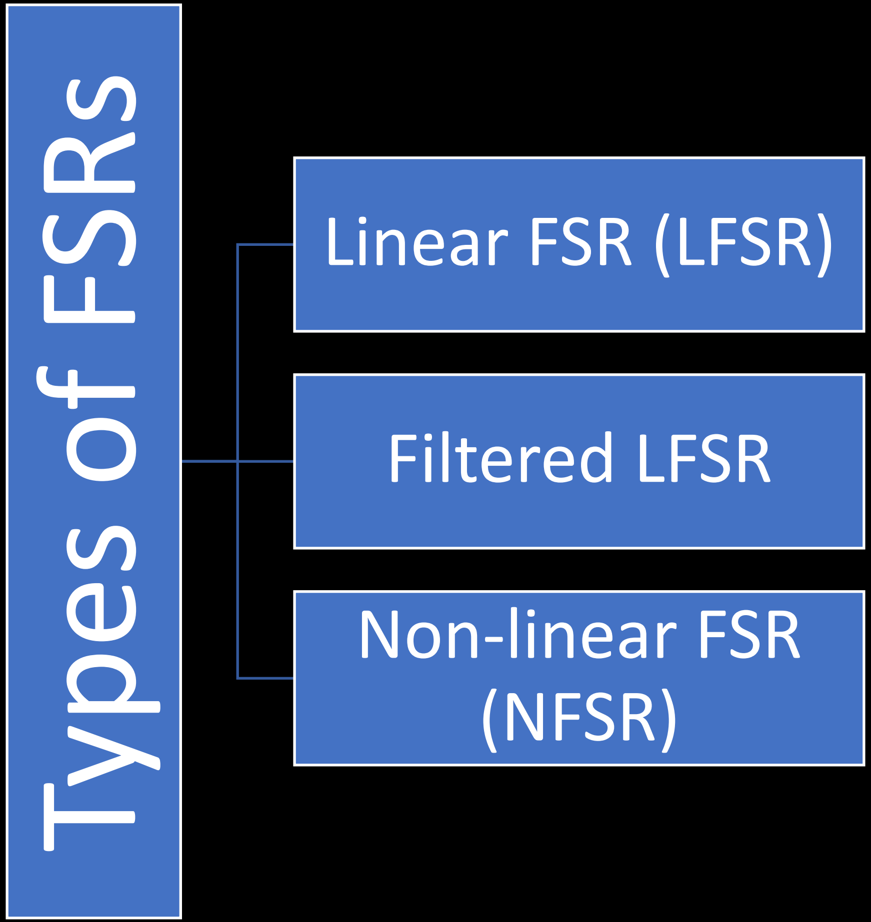




What is the period of 0000?



Hardware-Oriented Stream Ciphers



Hardware-Oriented Stream Ciphers

- LFSRs apply linear feedback function.
  - E.g., a function that do XOR of the bits of the state.

- In crypto, linear → patterns → predictable → bad.

- The choice of which bits are XORed together is crucial.
  - We want to guarantee a maximal period.
  - For an n-bit register, the maximal period is $2^n - 1$.

Hardware-Oriented Stream Ciphers

- LFSRs are represented as polynomials: $1 + X + X^2 + \cdots + X^n$.
  - The term X^i is the ith bit that is XORed in the feedback function.

- Example: the 4-bit LFSR is represented as $1 + X + X^3 + X^4$ because the bits 1, 3, and 4 are XORed in the feedback function.

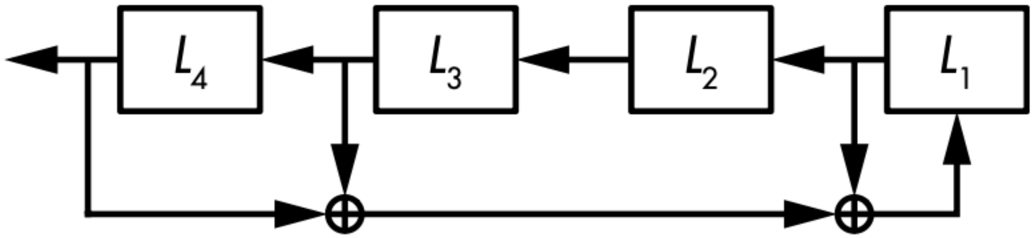


Figure 5-5: An LFSR with the feedback polynomial $1 + X + X^3 + X^4$

Hardware-Oriented Stream Ciphers

- The period of this LFSR is not maximal.
  - We get the initial state after six updates

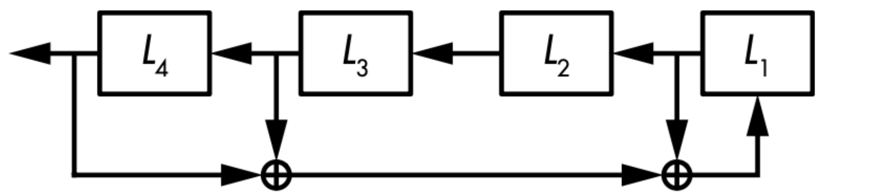


Figure 5-5: An LFSR with the feedback polynomial $1 + X + X^3 + X^4$

Initial state → 0 0 0 1
0 0 1 1
0 1 1 1
1 1 1 0
1 1 0 0
1 0 0 0
0 0 0 1

Hardware-Oriented Stream Ciphers

- This LFSR has maximal period
  - $1 + X^3 + X^4$
  - The initial value: 0 0 0 1

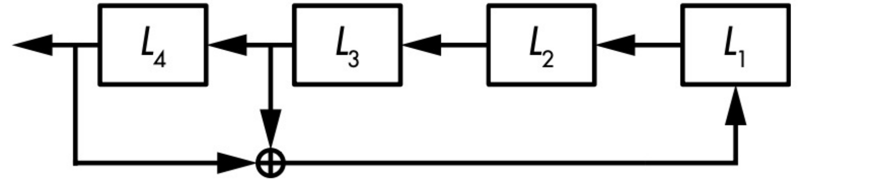
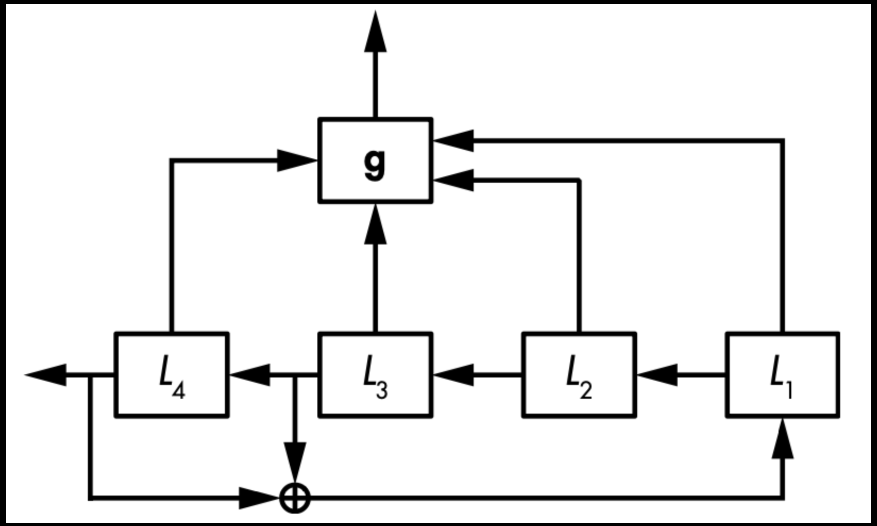


Figure 5-6: An LFSR with the feedback polynomial $1 + X^3 + X^4$, a primitive polynomial, ensuring a maximal period

0 0 0 1	0 0 1 1	0 1 0 1	1 1 1 0
0 0 1 0	0 1 1 0	1 0 1 1	1 1 0 0
0 1 0 0	1 1 0 1	0 1 1 1	1 0 0 0
1 0 0 1	1 0 1 0	1 1 1 1	0 0 0 1

Hardware-Oriented Stream Ciphers

- Filtered LFSR: hides the linearity of the LFSR by using nonlinear function.
  - The output of the LFSR is passed to a nonlinear function before producing the output



  - g is a nonlinear function that applies XORs with ANDs or ORs

Hardware-Oriented Stream Ciphers

- Complex attacks can break the filtered LFSR
  - *Patching a broken algorithm with a slightly stronger layer won't make the whole thing secure.*

- Attacks:
  - **Algebraic attacks:** solve the nonlinear equations deduced from the output bits.
  - **Cube attacks:** compute derivatives of the nonlinear equations to reduce the degree of the system down to one and then solve it.
  - **Fast correlation attacks:** exploit filtering functions that, despite their nonlinearity, tend to behave like linear functions.

Hardware-Oriented Stream Ciphers

- Nonlinear FSR: like LFSRs but with a nonlinear feedback function.
  - Uses ANDs, ORs and XORs

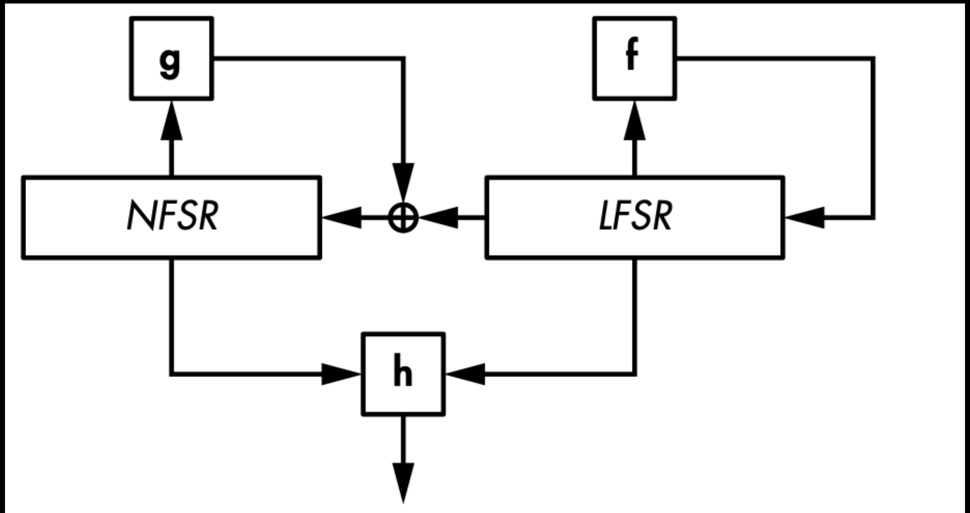
- Example: a 4-bit NFSR feedback function

$$N_1 + N_2 + N_1N_2 + N_3N_4$$

- **Downside:** cannot determine if its period is maximal
  - For n-bit NFSR, we need to run 2^n trials; impossible for large n (e.g., $n = 80$)

Hardware-Oriented Stream Ciphers

- Grain-128a: a stream cipher included in the eSTREAM 2008 competition.



  - Combines 128-bit LFSR, 128-bit NFSR, and a filter function h
  - The LFSR has a maximal period of at least $2^{128} - 1$

Hardware-Oriented Stream Ciphers

How Grain-128a works:

1. **Input**: 128-bit key and 96-bit nonce
    1. The 128-bit key goes to the NFSR, the 96-bit nonce goes to the LFSR
    2. The LFSR is padded with 31 ones and 1 zero bits
2. **Initialization phase:** updates the system 256 times before returning the first keystream bit.
3. **LFSR feedback function**: $\mathbf{f}(L) = L_{32} + L_{47} + L_{58} + L_{90} + L_{121} + L_{128}$
4. **NFSR feedback function**:

$$\mathbf{g}(N) = N_{32} + N_{37} + N_{72} + N_{102} + N_{128} + N_{44}N_{60} + N_{61}N_{125} + N_{63}N_{67} + N_{69}N_{101} + N_{80}N_{88} + N_{110}N_{111} + N_{115}N_{117} + N_{46}N_{50}N_{58} + N_{103}N_{104}N_{106} + N_{33}N_{35}N_{36}N_{40}$$

Hardware-Oriented Stream Ciphers

How Grain-128a works:

7. **Nonlinear h**: takes 9 bits from the NFSR and 7 bits from the LFSR and combines them.

- Used in low-level embedded systems:
    - Fast and secure
    - Proprietary

Hardware-Oriented Stream Ciphers

- **A5/1 cipher**: encrypts voice communications in the 2G mobile standard.
  - Broken
- It uses three LFSRs
- Key: 64 bits
- Nonce: 22 bits
  - Changed every time frame

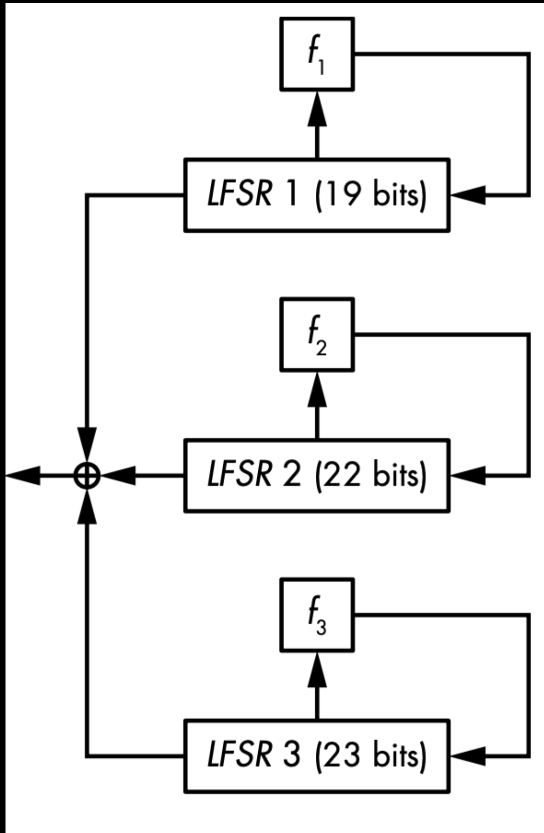
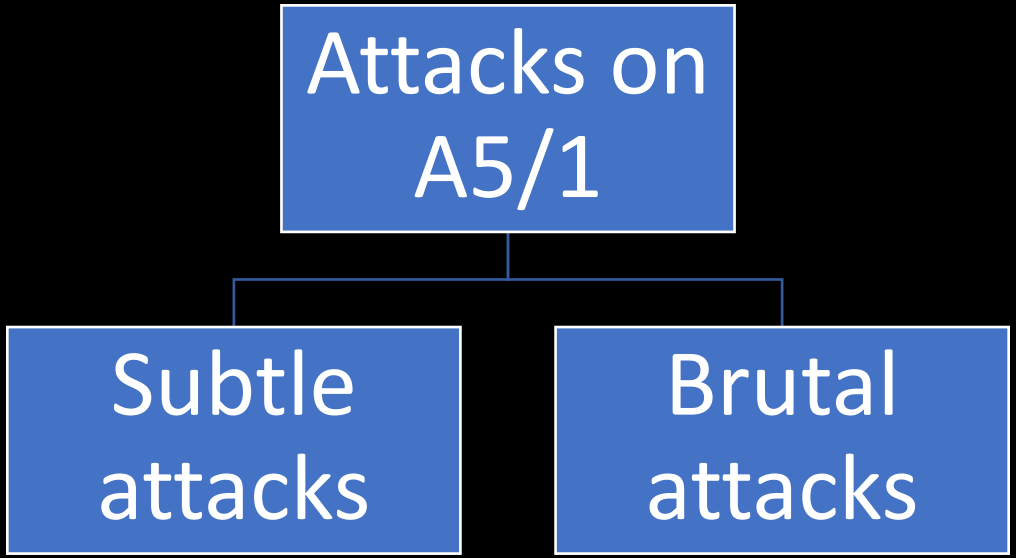


Figure 5-9: The A5/1 cipher

Hardware-Oriented Stream Ciphers



Hardware-Oriented Stream Ciphers

- **Subtle attacks:**
  - Called a *guess-and-determine* attack
  - An attacker guesses certain secret values of the state to determine others

```
For all 2^19 values of LFSR 1's initial state
    For all 2^22 values of LFSR 2's initial state
        For all 2^11 values of LFSR 3's clocking bit during the first 11 clocks
            Reconstruct LFSR 3's initial state
            Test whether guess is correct; if yes, return; else continue
```

  - This includes solving equations that depends on the bits guessed
  - Takes at most $2^{19} \times 2^{22} \times 2^{11} = 2^{52}$ operations instead of 2^{64}

Hardware-Oriented Stream Ciphers

- **Brutal attacks:**
  - Time-memory-trade-off (TMTO) attack
  - **Codebook attack** – precompute a table of 2^{64} of key-value pairs and store them
  - The attack is fast – just look up a value in memory
  - Takes very long to compute the table, requires large memory, 256 exabytes
  - **TMTO** attacks **reduce the memory required by a codebook** attack at the price of increased computation during the online phase of the attack
  - In 2010, researchers took about 2 months to generate two terabytes' tables, using GPUs and running 100,000 instances of A5/1 in parallel.

Hardware-Oriented Stream Ciphers

TASKS

1. Implement the LFSR class that takes an input an initial state and the bits position to XOR. Try with seed = [1, 0, 1, 1] and positions = [0, 2].
2. Implement a function that computes the period of the previous LFSR and its feedback function. What is the period of each of these feedback functions [0, 2, 3] and [0, 3] on the state [1, 0, 0, 0]?
3. Implement the NFSR class that takes an input an initial state and the bits position. The feedback bit is computed using: $N_2 + N_3$ and the output bit is computed using: $N_0 + N_1 + N_0 N_1 + N_2 N_3$

Content

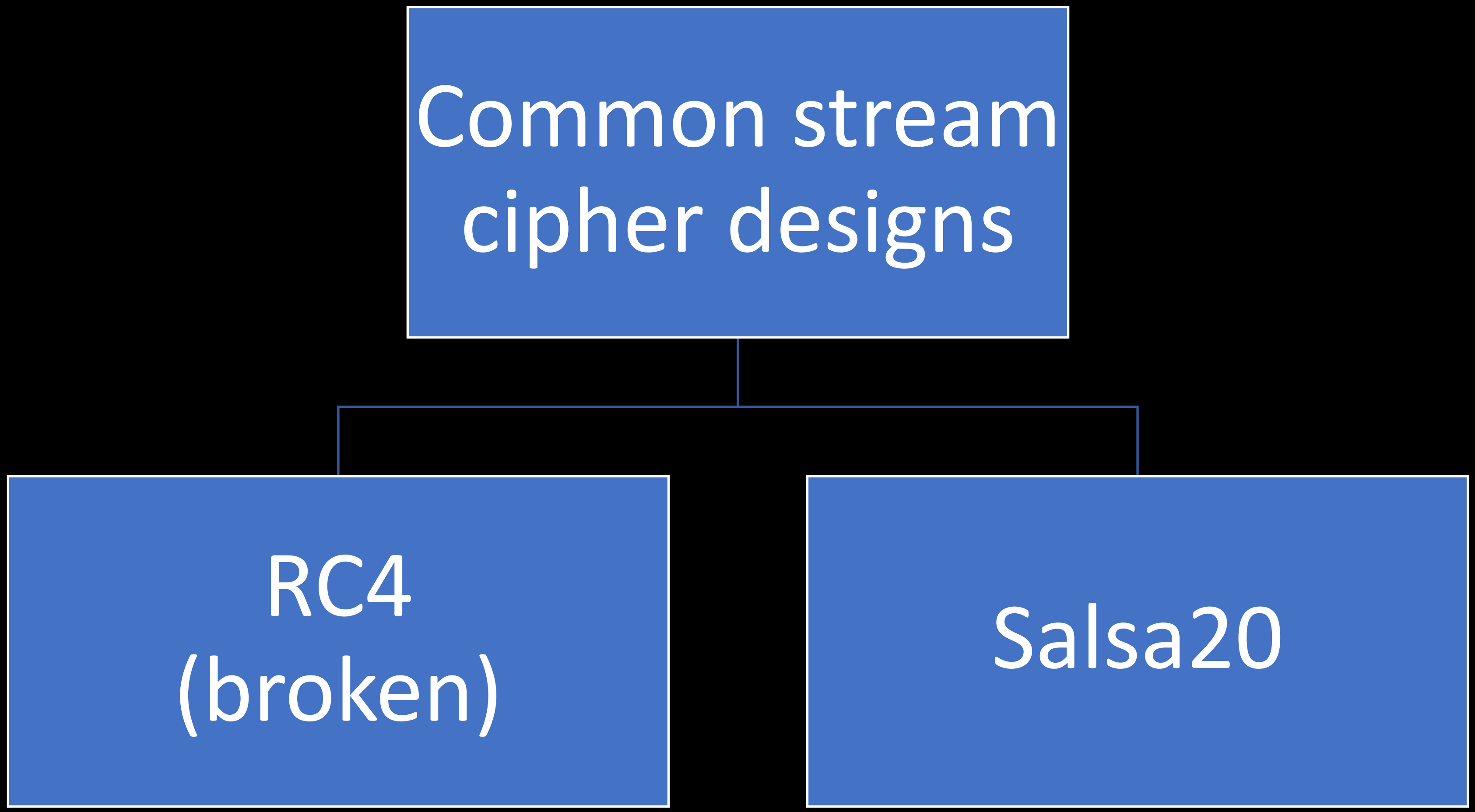
Content
Introduction
Hardware-Oriented Stream Ciphers
Software-Oriented Stream Ciphers

Software-Oriented Stream Ciphers

- Work with bytes, 32-bit, or 64-bit words instead of individual bits.
- Better than hardware ciphers when run as part of a software.
  - Browsers and servers

- Popularity of software stream ciphers:
  - Some 4G networks use stream ciphers that work with 32-bit words.
    - SONW3G and ZUC
  - Some block ciphers are weak against attacks
    - e.g., padding oracle attack against block ciphers in CBC mode
  - Easier to specify and to implement than block cipher
    - AES (block cipher) + CTR mode of operation → stream cipher

Software-Oriented Stream Ciphers



Software-Oriented Stream Ciphers
RC4

- RC4 was used in many applications:
  - WiFi encryption standard (WEP)
  - Transport Layer Security (TLS) protocol for establishing HTTPS connections.
- How RC4 works: Just swaps bytes (no complex operations)
  - It has an internal array, S, of size 256 bytes: $S[0] = 0, S[1] = 1, \dots, S[255] = 255$
  - S is initialized from a key, k, according to a key scheduling algorithm (KSA)

```
j = 0
# set S to the array S[0] = 0, S[1] = 1, . . . , S[255] = 255
S = range(256)
# iterate over i from 0 to 255
for i in range(256):
    # compute the sum of v
    j = (j + S[i] + K[i % n]) % 256
    # swap S[i] and S[j]
    S[i], S[j] = S[j], S[i]
```

Software-Oriented Stream Ciphers RC4

- The internal state S looks like this before executing the KSA:

$$0, 1, 2, 3, \dots, 255$$

- After executing the KSA, it will become

$$0, 35, 90, 116, \dots, 4$$

- If you change one bit of the key, you will get a totally different state

$$32, 11, 10,212, \dots, 2$$

Software-Oriented Stream Ciphers RC4

- After initializing the state S, RC4 generates the key stream KS.
  - Size(KS) = size(plaintext)
- The ciphertext is computed as $c = KS \oplus plaintext$
- The keystream is computed as
  - m is the size of the plaintext in bytes

```
i = 0
j = 0
for b in range(m):
    i = (i + 1) % 256
    j = (j + S[i]) % 256
    S[i], S[j] = S[j], S[i]
    KS[b] = S[(S[i] + S[j]) % 256]
```

Software-Oriented Stream Ciphers RC4

- RC4 in WEP

RC4 encrypts data of the wireless network

Software-Oriented Stream Ciphers
RC4

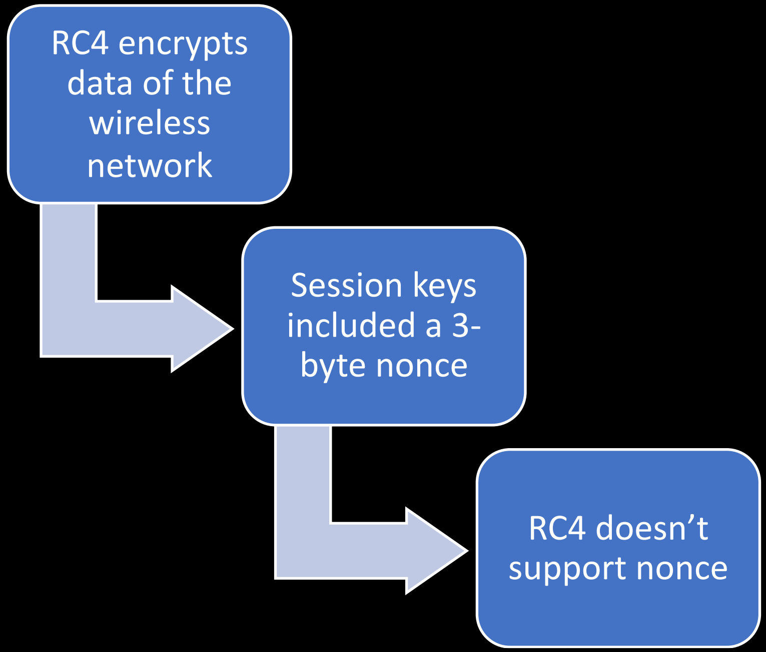
- RC4 in WEP

RC4 encrypts data of the wireless network

Session keys included a 3-byte nonce

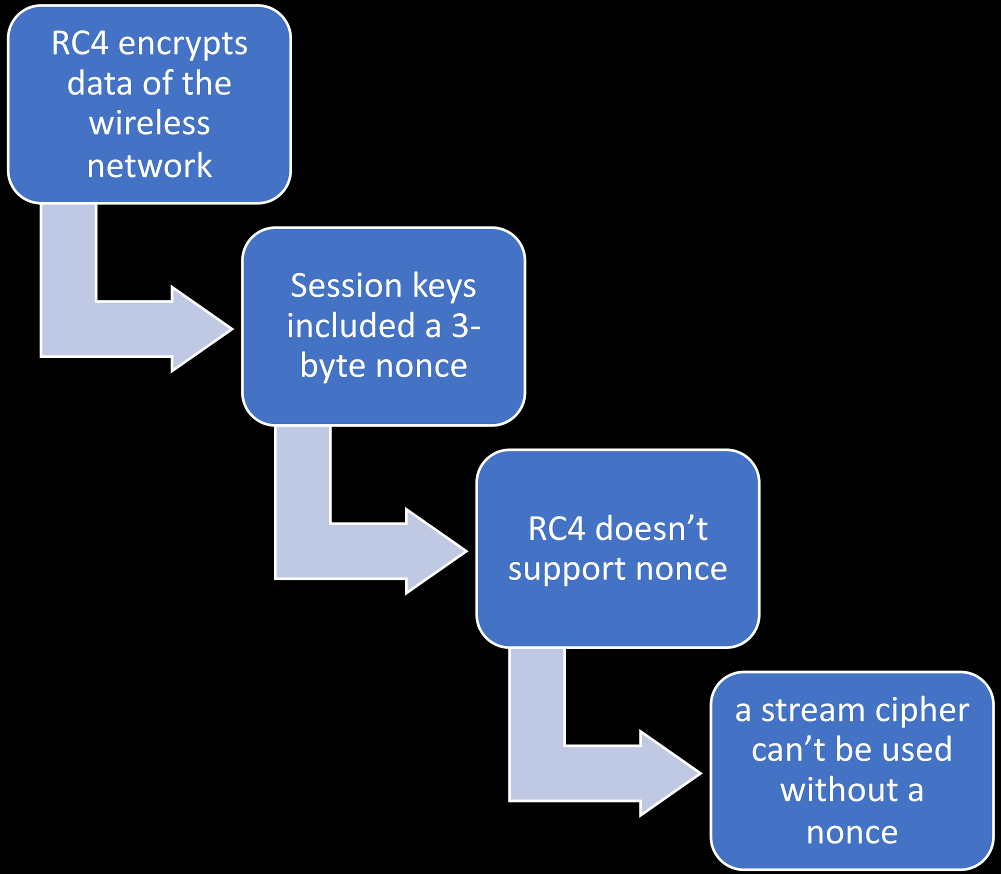
Software-Oriented Stream Ciphers
RC4

- RC4 in WEP



Software-Oriented Stream Ciphers RC4

- RC4 in WEP



Software-Oriented Stream Ciphers
RC4

- RC4 in WEP

RC4 encrypts data of the wireless network

Session keys included a 3-byte nonce

RC4 doesn't support nonce

a stream cipher can't be used without a nonce

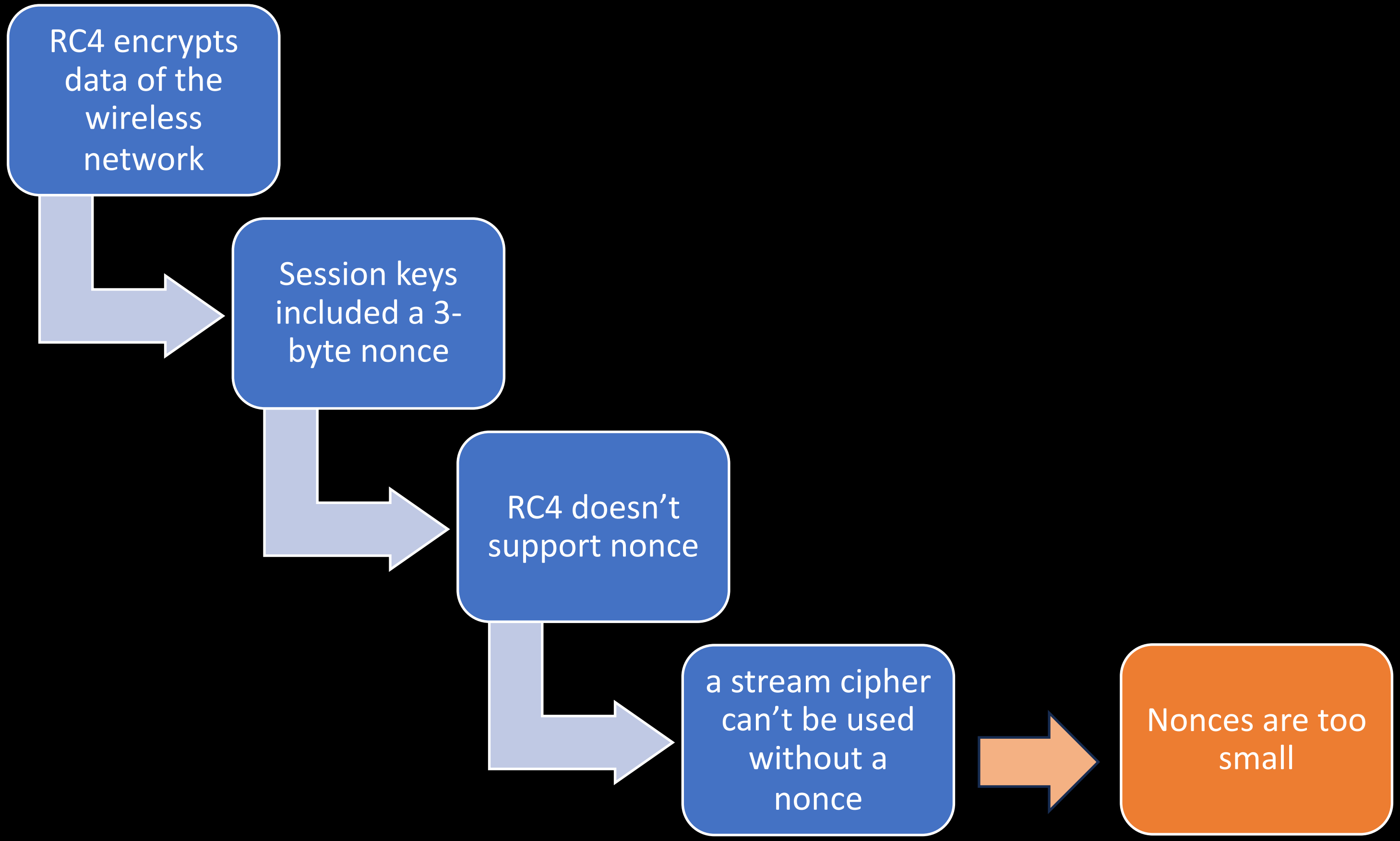
1. Include a 24-bit nonce in the wireless frame's header
2. Prepended it to the WEP key.

Ex: if the nonce is 123 and the key is ABC, then the RC4 key is 123ABC.

So, 40-bit keys, yield 64-bit security, and 104-bit keys, yield 128-bit key

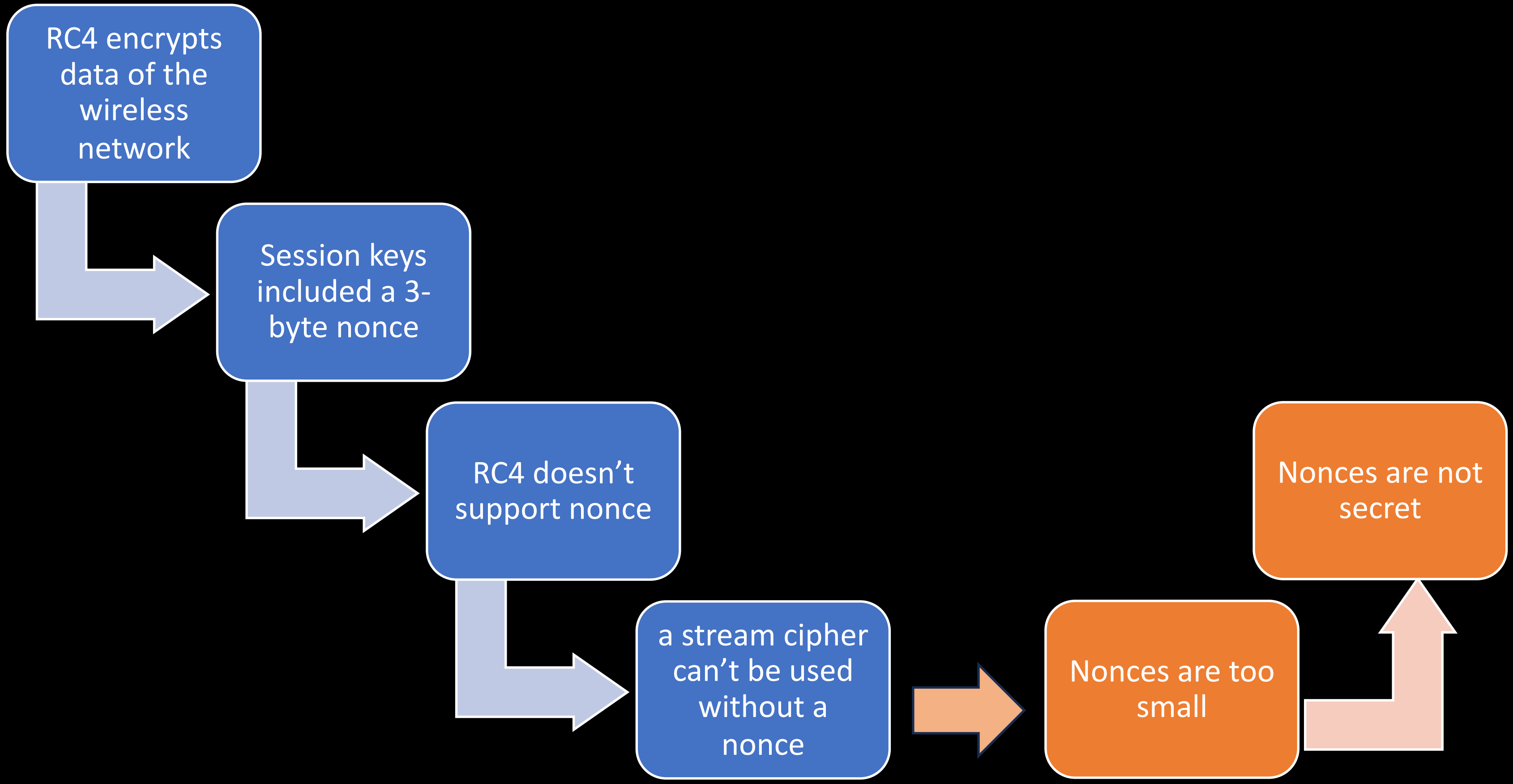
Software-Oriented Stream Ciphers RC4

- RC4 in WEP



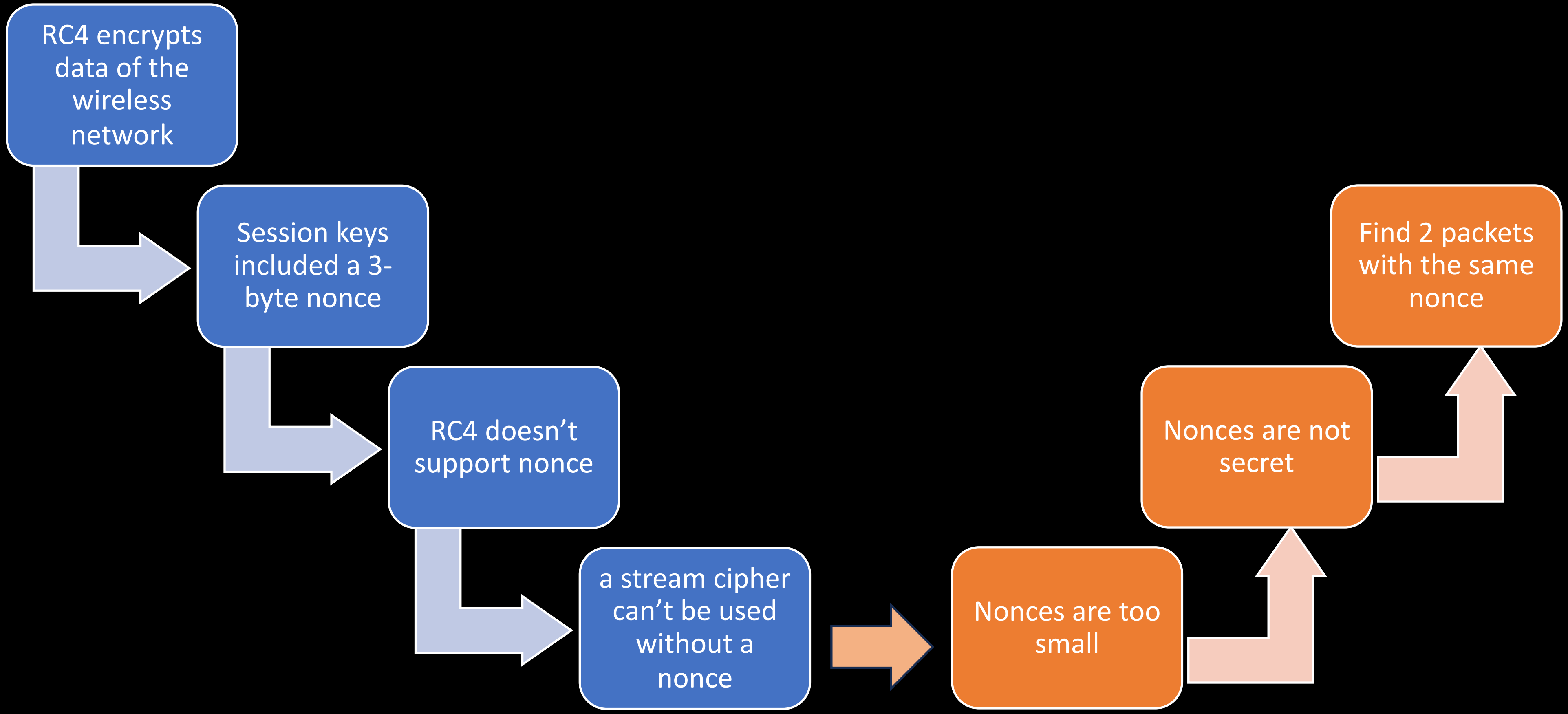
Software-Oriented Stream Ciphers RC4

- RC4 in WEP



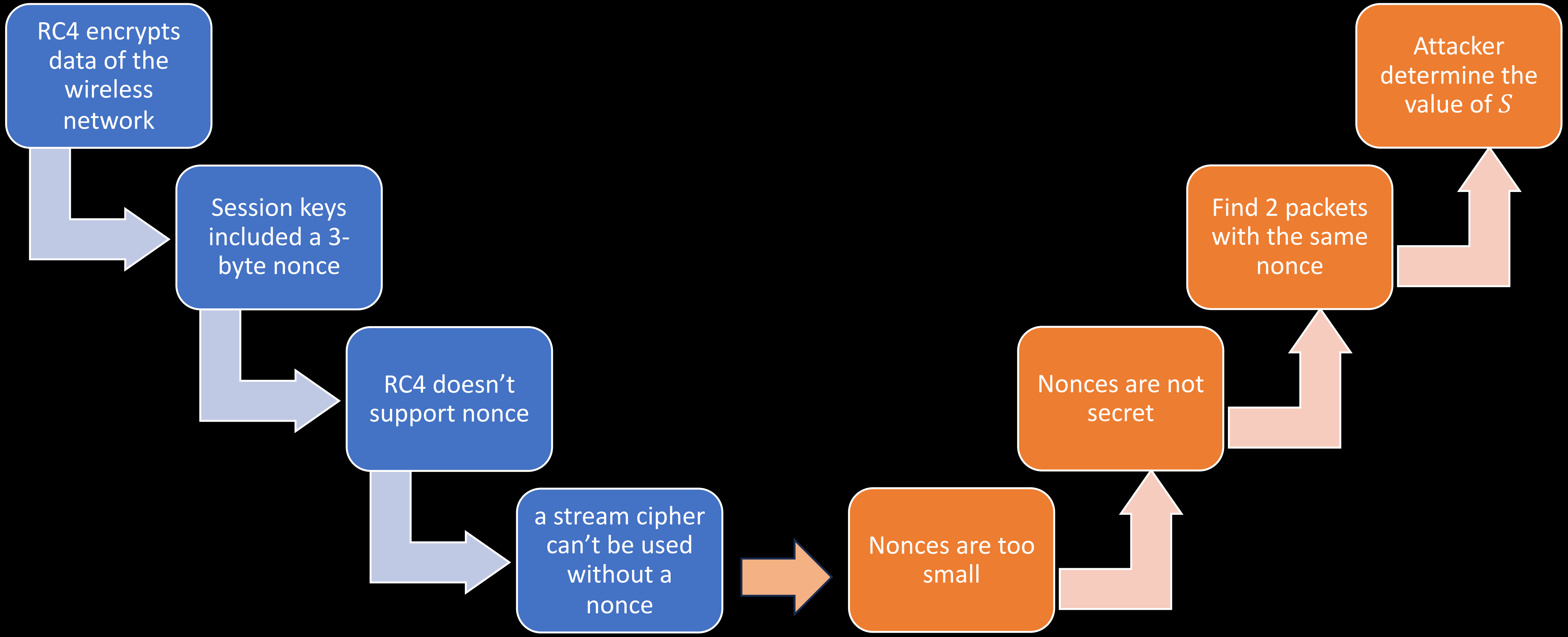
Software-Oriented Stream Ciphers RC4

- RC4 in WEP



Software-Oriented Stream Ciphers
RC4

- RC4 in WEP

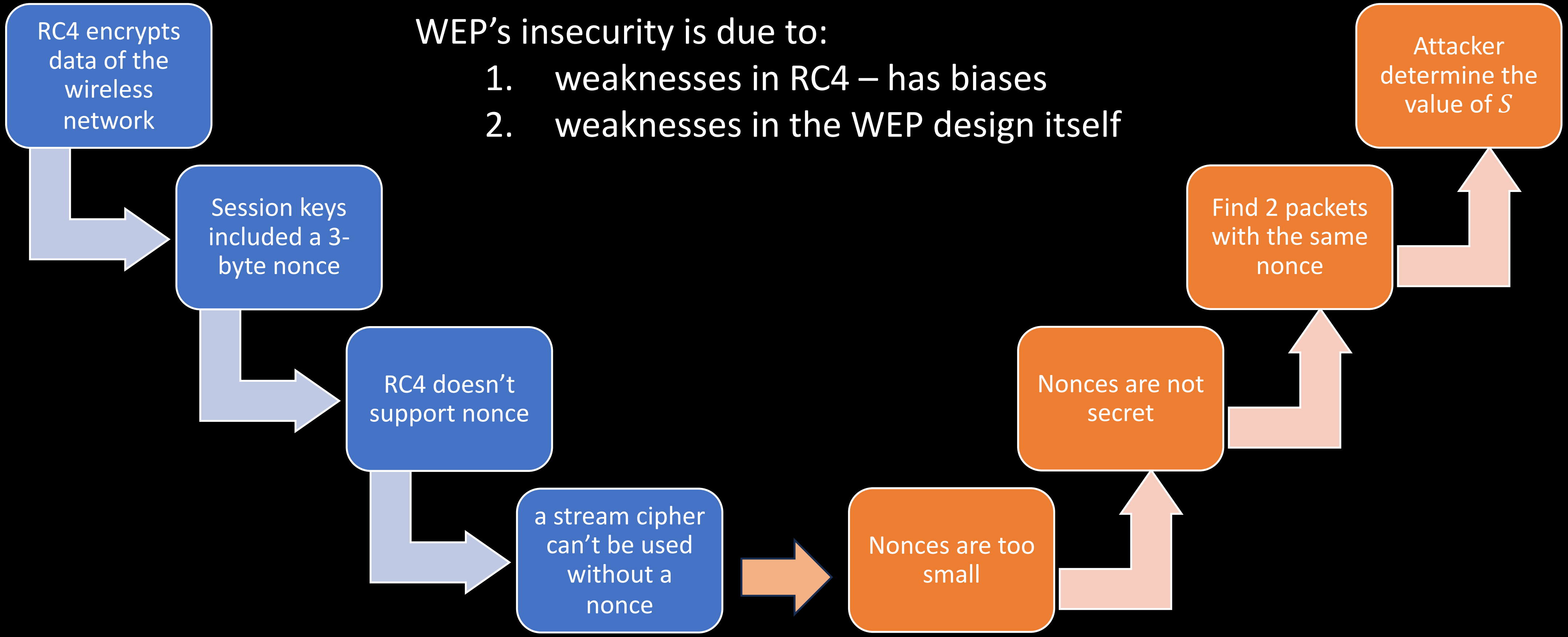


Software-Oriented Stream Ciphers
RC4

- RC4 in WEP

WEP's insecurity is due to:

1. weaknesses in RC4 – has biases
2. weaknesses in the WEP design itself



Software-Oriented Stream Ciphers
RC4

RC4 in TLS

- TLS inputs a unique 128-bit session key to the RC4.

- TLS was weak due only to RC4.
  - RC4 has statistical biases
  - Statistical biases are caused by non-randomness
  - Example: the 2^{nd} keystream byte is zero, with a probability of $\frac{1}{128}$, instead of $\frac{1}{256}$

Software-Oriented Stream Ciphers RC4

- RC4 in TLS attack: collect many ciphertext and just look for plaintext values
  - The ctxts should be encrypting the same ptxts <u>several times</u> using <u>different keys</u>.
  - This a *broadcast attack model*

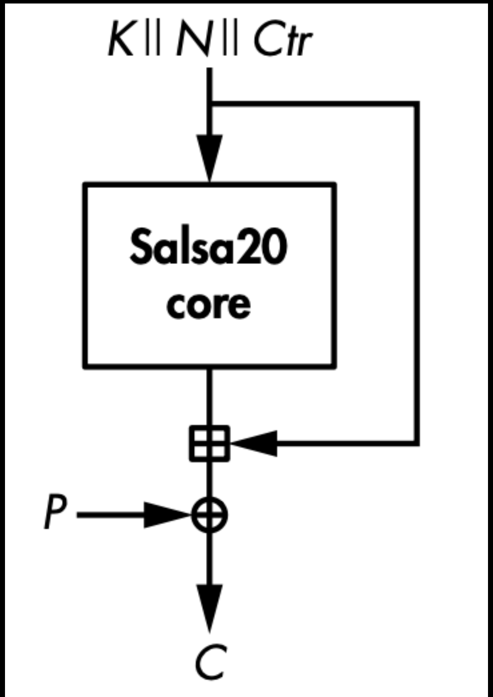
- This attack works because of RC4 is biased → the KS is likely to be zeros.
  - Therefore, some ciphertext bytes will be equal to the plaintext bytes.

- Hard to put the attack in practice
  - You need to trick the server to encrypt the same message several times.

Software-Oriented Stream Ciphers Salsa20

- Salsa20 is a counter-based stream cipher
  - It generates its keystream by processing a counter incremented for each block.

- Inputs: 512-bit block of key||nonce||counter
- Transforms the inputs using Salsa20 core function
- The original value of the block is added to the output of the Salsa20 core function to produce the keystream
- The keystream is XORed with the plaintext to get the ciphertext



Software-Oriented Stream Ciphers Salsa20

- Salsa20 core uses a *quarter round (QR)* permutation function.
- The QR transforms four 32-bit words a, b, c, d as follows:

$$
\begin{aligned}
b &= b \oplus \left[(a + d) <<< 7 \right] \\
c &= c \oplus \left[(b + a) <<< 9 \right] \\
d &= d \oplus \left[(c + b) <<< 13 \right] \\
a &= a \oplus \left[(d + c) <<< 18 \right]
\end{aligned}
$$

Software-Oriented Stream Ciphers Salsa20

- 512-bit block transformation → apply QR to a 4x4 state of 32-bit words.

- The 4x4 internal state consists of:
  - Key: 8 words (256 bits), k
  - Nonce: 2 words (64 bits), v
  - Counter: 2 words (64 bits), t
  - Constants: 4 words (128 bits), c

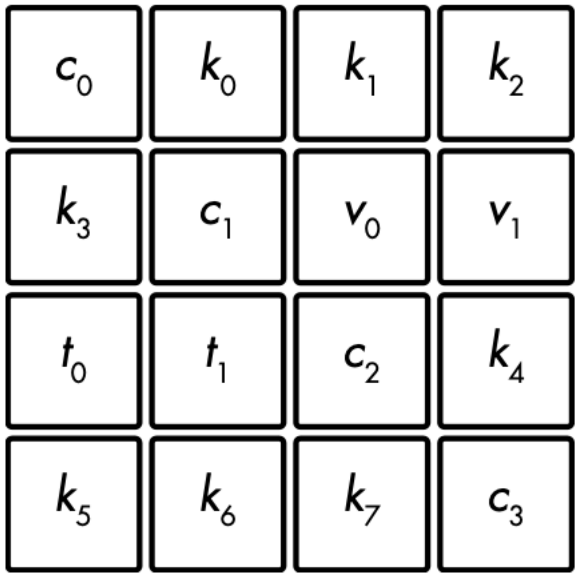


Figure 5-11: The initialization of Salsa20's state

Software-Oriented Stream Ciphers Salsa20

- Salsa20 applies the QR to the columns and the rows of the internal state.
  - **Row round**: transforming all the 4 rows independently
  - **Column round**: transforming all the 4 columns independently
  - The sequence column-round/row-round is called a **double-round**.
- It repeats 10 double-rounds, for 20 rounds in total, thus the 20 in Salsa20.

$$\begin{aligned}
&\mathbf{QR}(x_0, x_1, x_2, x_3)\\
&\mathbf{QR}(x_5, x_6, x_7, x_4)\\
&\mathbf{QR}(x_{10}, x_{11}, x_8, x_9)\\
&\mathbf{QR}(x_{15}, x_{12}, x_{13}, x_{14})
\end{aligned}$$

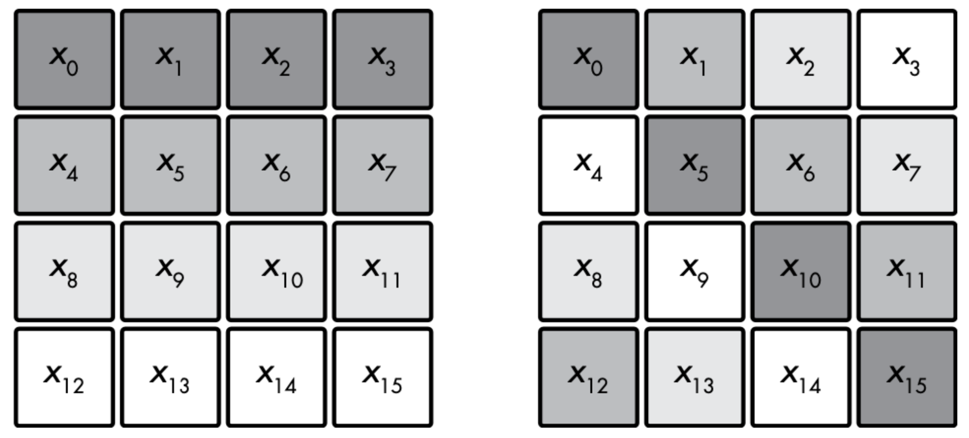


*Figure 5-12: Columns and rows transformed by Salsa20's quarter-round (**QR**) function*

$$\begin{aligned}
&\mathbf{QR}(x_0, x_4, x_8, x_{12})\\
&\mathbf{QR}(x_1, x_5, x_9, x_{13})\\
&\mathbf{QR}(x_2, x_6, x_{10}, x_{14})\\
&\mathbf{QR}(x_3, x_7, x_{11}, x_{15})
\end{aligned}$$

Software-Oriented Stream Ciphers Salsa20

- To show why Salsa20 is more secure than RC4, use **differential cryptanalysis**
  - The study of the differences between states rather than their actual values.
- Example: given two states of the Salsa20
  - Both states have the same key (all zeros), nonce (all ff), and constants
  - The first state has counter 0 and the second block has counter one

61707865	00000000	00000000	00000000
00000000	3320646e	ffffffff	ffffffff
00000000	00000000	79622d32	00000000
00000000	00000000	00000000	6b206574

61707865	00000000	00000000	00000000
00000000	3320646e	ffffffff	ffffffff
00000001	00000000	79622d32	00000000
00000000	00000000	00000000	6b206574

Listing 5-3: Salsa20's initial states for the first two blocks with an all-zero key and an all-one nonce

Software-Oriented Stream Ciphers
Salsa20

Only one bit difference between the states

61707865	00000000	00000000	00000000	61707865	00000000	00000000	00000000
00000000	3320646e	ffffffff	ffffffff	00000000	3320646e	ffffffff	ffffffff
00000000	00000000	79622d32	00000000	**00000001**	00000000	79622d32	00000000
00000000	00000000	00000000	6b206574	00000000	00000000	00000000	6b206574

After 10 double rounds, everything looks random and no statistical bias

*In RC4, everything look random, but **had statistical biases***

e98680bc	f730ba7a	38663ce0	5f376d93	1ba4d492	c14270c3	9fb05306	ff808c64
85683b75	a56ca873	26501592	64144b6d	b49a4100	f5d8fbbd	614234a0	e20663d1
6dcb46fd	58178f93	8cf54cfe	cfdc27d7	12e1e116	6a61bc8f	86f01bcb	2efead4a
68bbe09e	17b403a1	38aa1f27	54323fe0	77775a13	d17b99d5	eb773f5b	2c3a5e7d

Listing 5-4: The states from Listing 5-3 after 10 Salsa20 double-rounds

Software-Oriented Stream Ciphers
Salsa20

How this works?

If you compute the difference (XOR) between these two states,

61707865	00000000	00000000	00000000		61707865	00000000	00000000	00000000
00000000	3320646e	ffffffff	ffffffff		00000000	3320646e	ffffffff	ffffffff
00000000	00000000	79622d32	00000000		**00000001**	00000000	79622d32	00000000
00000000	00000000	00000000	6b206574		00000000	00000000	00000000	6b206574

You will get this state

00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000001	00000000	00000000	00000000
00000000	00000000	00000000	00000000

Software-Oriented Stream Ciphers
Salsa20

How this works?

00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000001	00000000	00000000	00000000
00000000	00000000	00000000	00000000

Software-Oriented Stream Ciphers
Salsa20

How this works?

00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000001	00000000	00000000	00000000
00000000	00000000	00000000	00000000

After 1 round

80040003	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000001	00000000	00000000	00000000
00002000	00000000	00000000	00000000

Software-Oriented Stream Ciphers
Salsa20

How this works?

00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000001	00000000	00000000	00000000
00000000	00000000	00000000	00000000

After 1 round

80040003	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000001	00000000	00000000	00000000
00002000	00000000	00000000	00000000

After 1 round

9ed7eb7f	**060002c0**	**18028b0c**	**57ca83c0**
00000000	00000000	00000000	00000000
00000001	**0000e000**	**801c0006**	00000000
00002000	**00400000**	**04000008**	**0060f300**

Software-Oriented Stream Ciphers
Salsa20

How this works?

00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000001	00000000	00000000	00000000
00000000	00000000	00000000	00000000

After 1 round

80040003	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000001	00000000	00000000	00000000
00002000	00000000	00000000	00000000

After 1 round

9ed7eb7f	**060002c0**	**18028b0c**	**57ca83c0**
00000000	00000000	00000000	00000000
00000001	**0000e000**	**801c0006**	00000000
00002000	**00400000**	**04000008**	**0060f300**

After 1 round

3ab3c25d	9f40a5c9	10070e30	07bd03c0
db1ee2ce	43ee9401	21a702c3	48fd800c
403c1e72	00034003	4dc843be	700b8857
5625b75b	09c00e00	06000348	23f712d4

Software-Oriented Stream Ciphers
Salsa20

How this works?

00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000001	00000000	00000000	00000000
00000000	00000000	00000000	0000[illegible]

After 1 round

80040003	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000001	00000000	00000000	00000000
00002000	00000000	00000000	00000000

After 1 round

9ed7eb7f	**060002c0**	**18028b0c**	**57ca83c0**
00000000	00000000	00000000	00000000
00000001	**0000e000**	**801c0006**	00000000
00002000	**00400000**	**04000008**	**0060f300**

After 1 round

3ab3c25d	9f40a5c9	10070e30	07bd03c0
db1ee2ce	43ee9401	21a702c3	48fd800c
403c1e72	00034003	4dc843be	700b8857
[illegible]b75b	09c00e00	06000348	23f712d4

After 1 round

d93bed6d	a267bf47	760c2f9f	4a41d54b
0e03d792	7340e010	119e6a00	e90186af
7fa9617e	b6aca0d7	4f6e9a4a	564b34fd
98be796d	64908d32	4897f7ca	a684a2df

Software-Oriented Stream Ciphers
Salsa20

So, after only 4 rounds, a single difference propagates to the 512-bit state.

00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000001	00000000	00000000	00000000
00000000	00000000	00000000	00000000

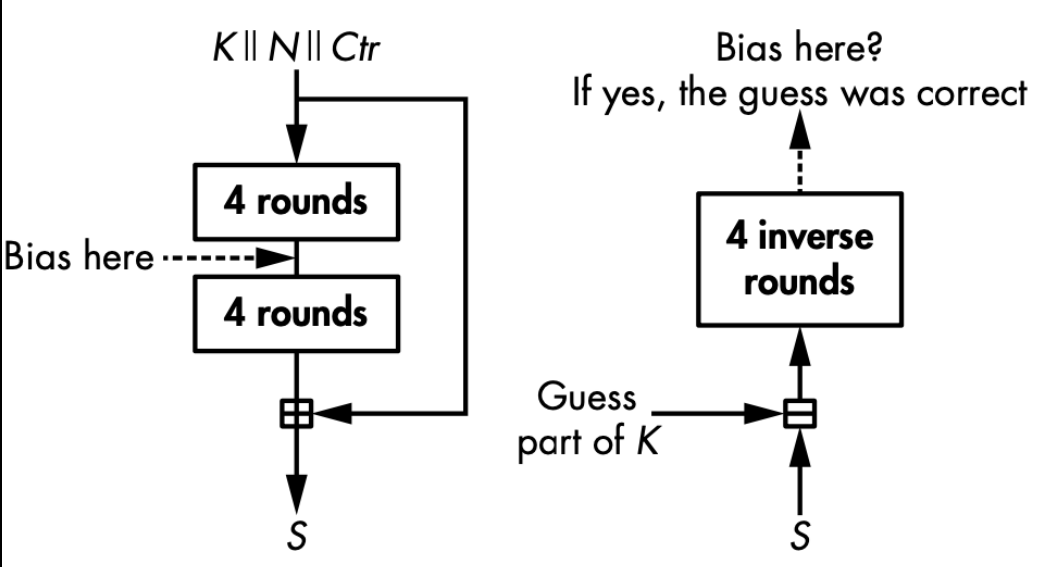
After 4 rounds →

d93bed6d	a267bf47	760c2f9f	4a41d54b
0e03d792	7340e010	119e6a00	e90186af
7fa9617e	b6aca0d7	4f6e9a4a	564b34fd
98be796d	64908d32	4897f7ca	a684a2df

This is called **full diffusion**

Software-Oriented Stream Ciphers
Salsa20

- **Attack on Salsa20/8:** exploit a statistical bias in the core algorithm after 4 rounds
    - Salsa20/8 is a smaller version of Salsa20 that uses 8 rounds instead of 20.
    - Published in *New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba*
    - Theoretical attack: its complexity is 2^{251} instead of 2^{256}



Software-Oriented Stream Ciphers Salsa20

TASKS

1. Use $pycryptodome$ to encrypt a message with $Salsa20$ and $ChaCha20$ ciphers.

Nonce length	Description	Max data	If random nonce and same key
8 bytes (default)	The original ChaCha20 designed by Bernstein.	No limitations	Max 200 000 messages
12 bytes	The TLS ChaCha20 as defined in RFC7539.	256 GB	Max 13 billions messages
24 bytes	XChaCha20, still in draft stage.	256 GB	No limitations

Software-Oriented Stream Ciphers
Salsa20

1. Implement a stateful-stream-cipher as follows:
   - The $init$ function calls a $secure\_PRP$ function and returns a $state$
   - The $secure\_PRP$ function takes a key and a nonce and returns the encryption of the nonce (the initial state). The PRP function is the AES cipher in ECB mode.
   - The $update$ function takes the $previous\_state$ as a parameter and calls the $secure\_PRP$ function to encrypt the $previous\_state$. The key is 16-bytes 0s.
   - Implement the encryption and decryption. The block size is 16 bytes (128 bits)
   - What happens if we set the AES mode of operation to CBC mode?