

How crypto fails in practice? CSS, WEP, MIFARE classic

*Slides borrowed from Vitaly Shmatikov

Stream Ciphers

◆ One-time pad:

$\text{Ciphertext}(\text{Key}, \text{Message}) = \text{Message} \oplus \text{Key}$

- Key must be a random bit sequence as long as message

◆ Idea: replace “random” with “pseudo-random”

- Use a pseudo-random number generator (PRNG)
- PRNG takes a short, truly random secret seed and expands it into a long “random-looking” sequence
 - E.g., 128-bit seed into a 10^6 -bit pseudo-random sequence

No efficient algorithm can tell this sequence from truly random

◆ $\text{Ciphertext}(\text{Key}, \text{Msg}) = \text{IV}, \text{Msg} \oplus \text{PRNG}(\text{IV}, \text{Key})$

- Message processed bit by bit (unlike block cipher)

Stream Cipher Terminology

- ◆ The seed of a pseudo-random generator typically consists of **initialization vector (IV)** and **key**
 - The key is a secret known only to the sender and the recipient, not sent with the ciphertext
 - IV is usually sent with the ciphertext
- ◆ The pseudo-random bit stream produced by $\text{PRNG}(\text{IV}, \text{key})$ is referred to as the **keystream**
- ◆ Encrypt message by XORing with keystream
 - $\text{ciphertext} = \text{message} \oplus \text{keystream}$

Properties of Stream Ciphers

- ◆ Usually very fast (faster than block ciphers)
 - Used where speed is important: WiFi, DVD, RFID, VoIP
- ◆ Unlike one-time pad, stream ciphers do not provide perfect secrecy
 - Only as secure as the underlying PRNG
 - If used properly, can be as secure as block ciphers
- ◆ PRNG must be cryptographically secure

Using Stream Ciphers

◆ No integrity

- Associativity & commutativity:

$$(M_1 \oplus \text{PRNG}(\text{seed})) \oplus M_2 = (M_1 \oplus M_2) \oplus \text{PRNG}(\text{seed})$$

- Need an additional integrity protection mechanism

◆ Known-plaintext attack is very dangerous if keystream is ever repeated

- Self-cancellation property of XOR: $X \oplus X = 0$
- $(M_1 \oplus \text{PRNG}(\text{seed})) \oplus (M_2 \oplus \text{PRNG}(\text{seed})) = M_1 \oplus M_2$
- If attacker knows M_1 , then easily recovers M_2 ...
also, most plaintexts contain enough redundancy that can recover parts of both messages from $M_1 \oplus M_2$

How Random is "Random"?

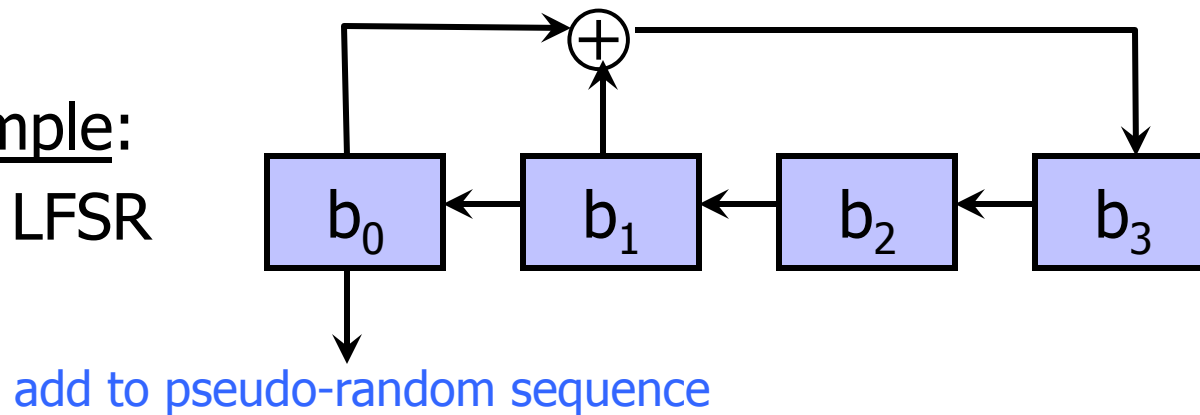


Cryptographically Secure PRNG

- ◆ Next-bit test: given N bits of the pseudo-random sequence, predict $(N+1)^{\text{st}}$ bit
 - Probability of correct prediction should be very close to $1/2$ for any efficient adversarial algorithm
(means what?)
- ◆ PRNG state compromise
 - Even if the attacker learns the complete or partial state of the PRNG, he should not be able to reproduce the previously generated sequence
 - ... or future sequence, if there' ll be future random seed(s)
- ◆ Common PRNGs are not cryptographically secure

LFSR: Linear Feedback Shift Register

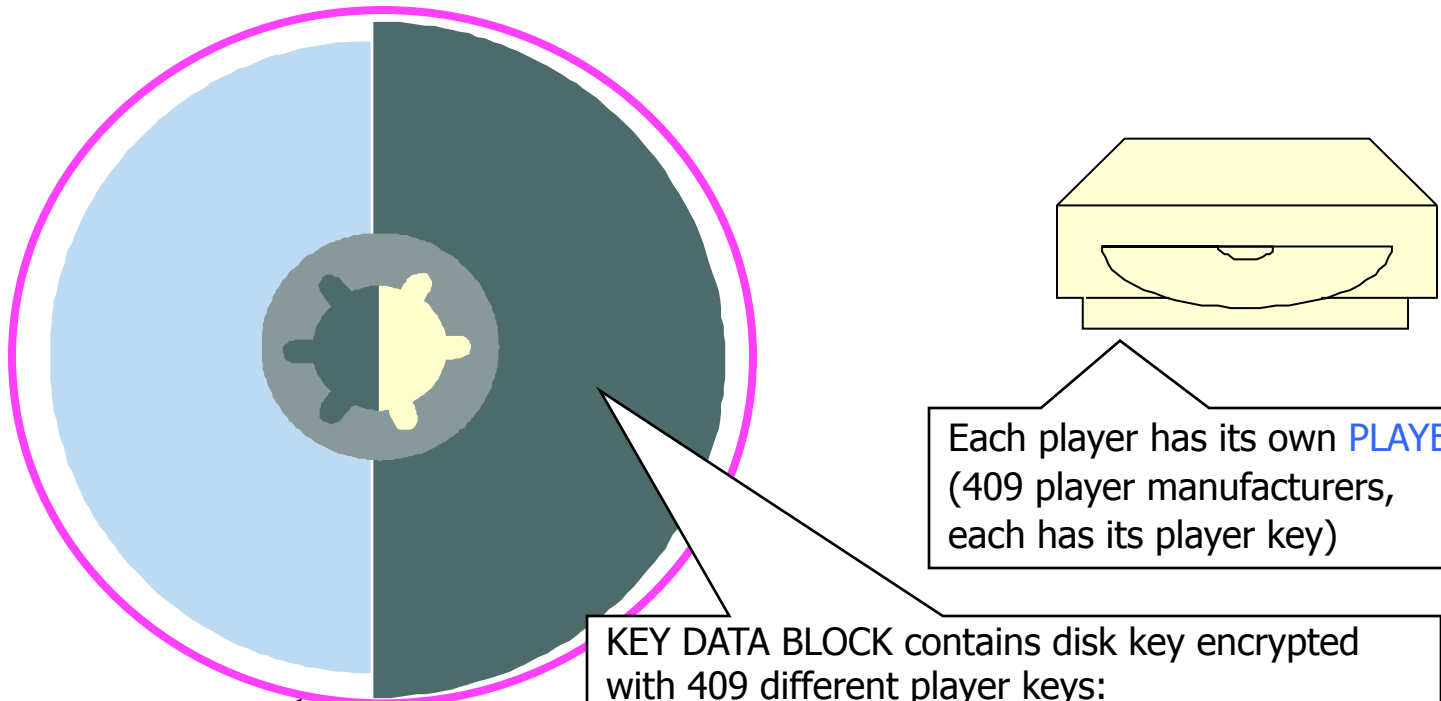
Example:
4-bit LFSR



- ◆ For example, if the seed is 1001, the generated sequence is 1001101011110001001...
- ◆ Repeats after 15 bits (2^4-1)

Content Scrambling System (CSS)

- ◆ DVD encryption scheme from Matsushita and Toshiba

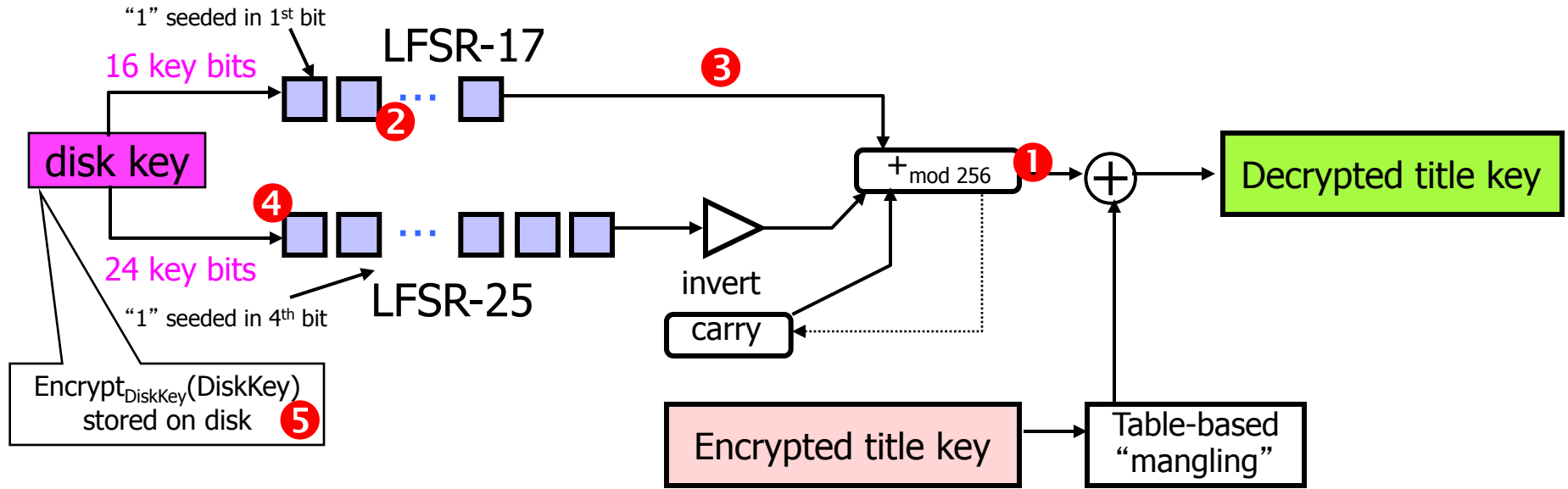


This helps attacker verify his guess of disk key

What happens if even a single player key is compromised?

Attack on CSS Decryption Scheme

[Frank Stevenson]



- Given known 40-bit plaintext, repeat the following 5 times (once for each plaintext byte): guess the byte output by the sum of the two LFSRs; use known ciphertext to verify – this takes $O(2^8)$
 - For each guessed output byte, guess 16 bits contained in LFSR-17 – this takes $O(2^{16})$
 - Clock out 24 bits out of LFSR-17, use subtraction to determine the corresponding output bits of LFSR-25 – this reveals all of LFSR-25 except the highest bit
 - “Roll back” 24 bits, try both possibilities – this takes $O(2)$
 - Clock out 16 more bits out of both LFSRs, verify the key
- This attack takes $O(2^{25})$**

DeCSS

- ◆ In CSS, disk key is encrypted under hundreds of different player keys... including Xing, a software DVD player
- ◆ Reverse engineering the object code of Xing revealed its player key
 - Every CSS disk contains the master disk key encrypted under Xing's key
 - One bad player \Rightarrow entire system is broken!
- ◆ Easy-to-use DeCSS software

DeCSS Aftermath

- ◆ DVD CCA sued Jon Lech Johansen (“DVD Jon”), one of DeCSS authors - eventually dropped
- ◆ Publishing DeCSS code violates copyright
 - Underground distribution as haikus and T-shirts
 - “Court to address DeCSS T-Shirt: When can a T-shirt become a trade secret? When it tells you how to copy a DVD.” - Wired News



RC4

- ◆ Designed by Ron Rivest for RSA in 1987
- ◆ Simple, fast, widely used
 - SSL/TLS for Web security, WEP for wireless

Byte array $S[256]$ contains a permutation of numbers from 0 to 255

$i = j := 0$

loop

$i := (i+1) \bmod 256$

$j := (j+S[i]) \bmod 256$

swap($S[i], S[j]$)

output $(S[i]+S[j]) \bmod 256$

end loop

RC4 Initialization

Divide key K into L bytes

Key can be any length
up to 2048 bits

for i = 0 to 255 do

 S[i] := i

j := 0

for i = 0 to 255 do

 j := (j+S[i]+K[i mod L]) mod 256

Generate initial permutation
from key K

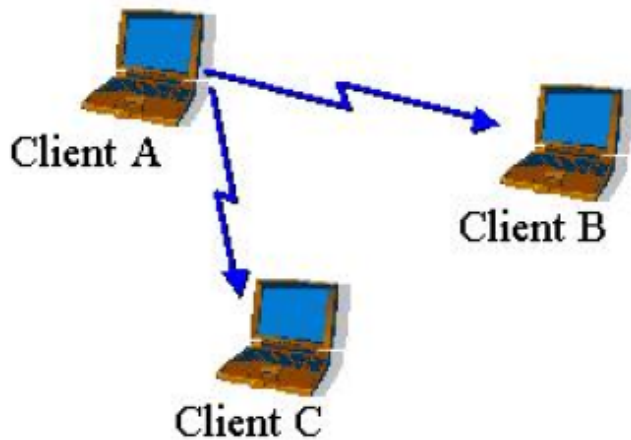
 swap(S[i],S[j])

- ◆ To use RC4, usually prepend **initialization vector** (IV) to the key
 - IV can be random or a counter
- ◆ RC4 is not random enough... First byte of generated sequence depends only on 3 cells of state array S - this can be used to extract the key!
 - To use RC4 securely, RSA suggests discarding first 256 bytes

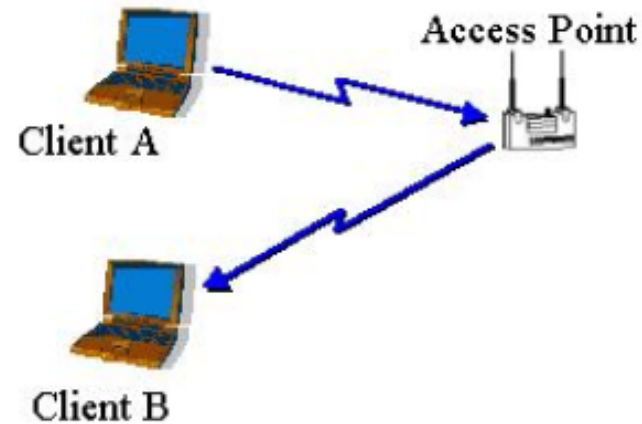
Fluhrer-Mantin-Shamir attack

802.11b Overview

- ◆ Standard for wireless networks (IEEE 1999)
- ◆ Two modes: **infrastructure** and **ad hoc**



IBSS (ad hoc) mode



BSS (infrastructure) mode

Access Point SSID

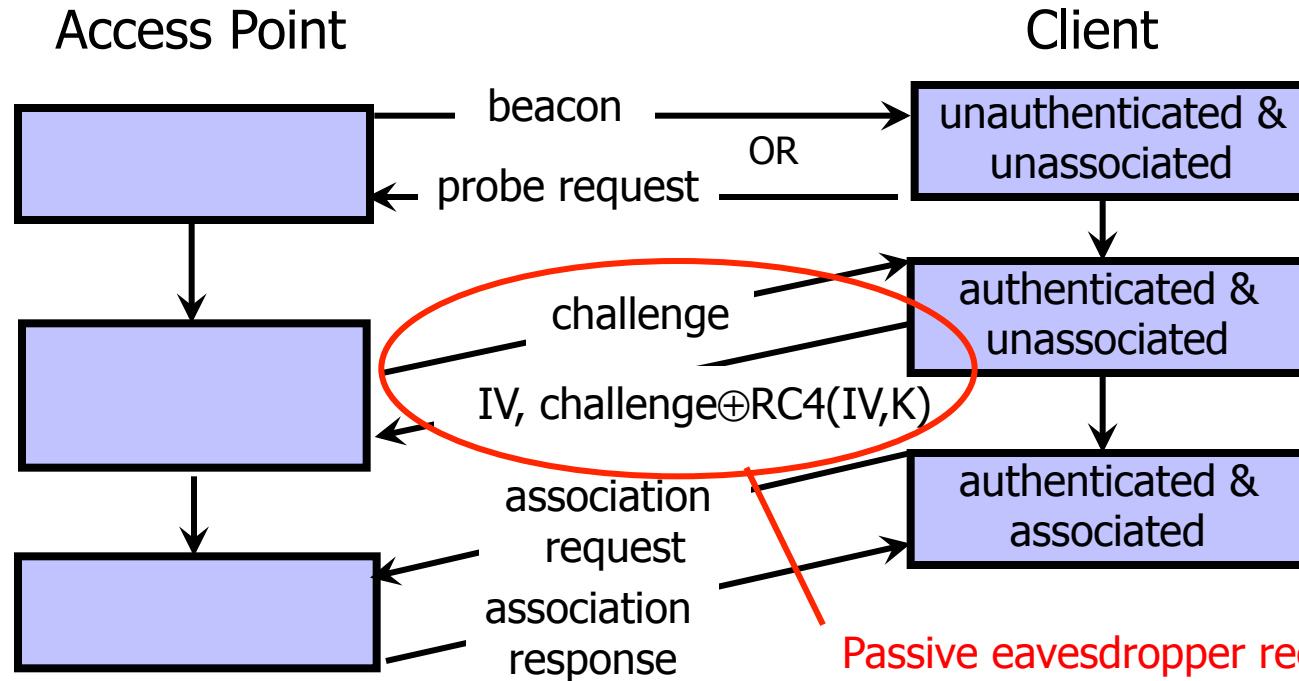
- ◆ Service Set Identifier (SSID) is the “name” of the access point
 - By default, access point broadcasts its SSID in plaintext “beacon frames” every few seconds
- ◆ Default SSIDs are easily guessable
 - Manufacturer’s defaults: “linksys”, “tsunami”, etc.
 - This gives away the fact that access point is active
- ◆ Access point settings can be changed to prevent it from announcing its presence in beacon frames and from using an easily guessable SSID
 - But then every user must know SSID in advance

WEP: Wired Equivalent Privacy

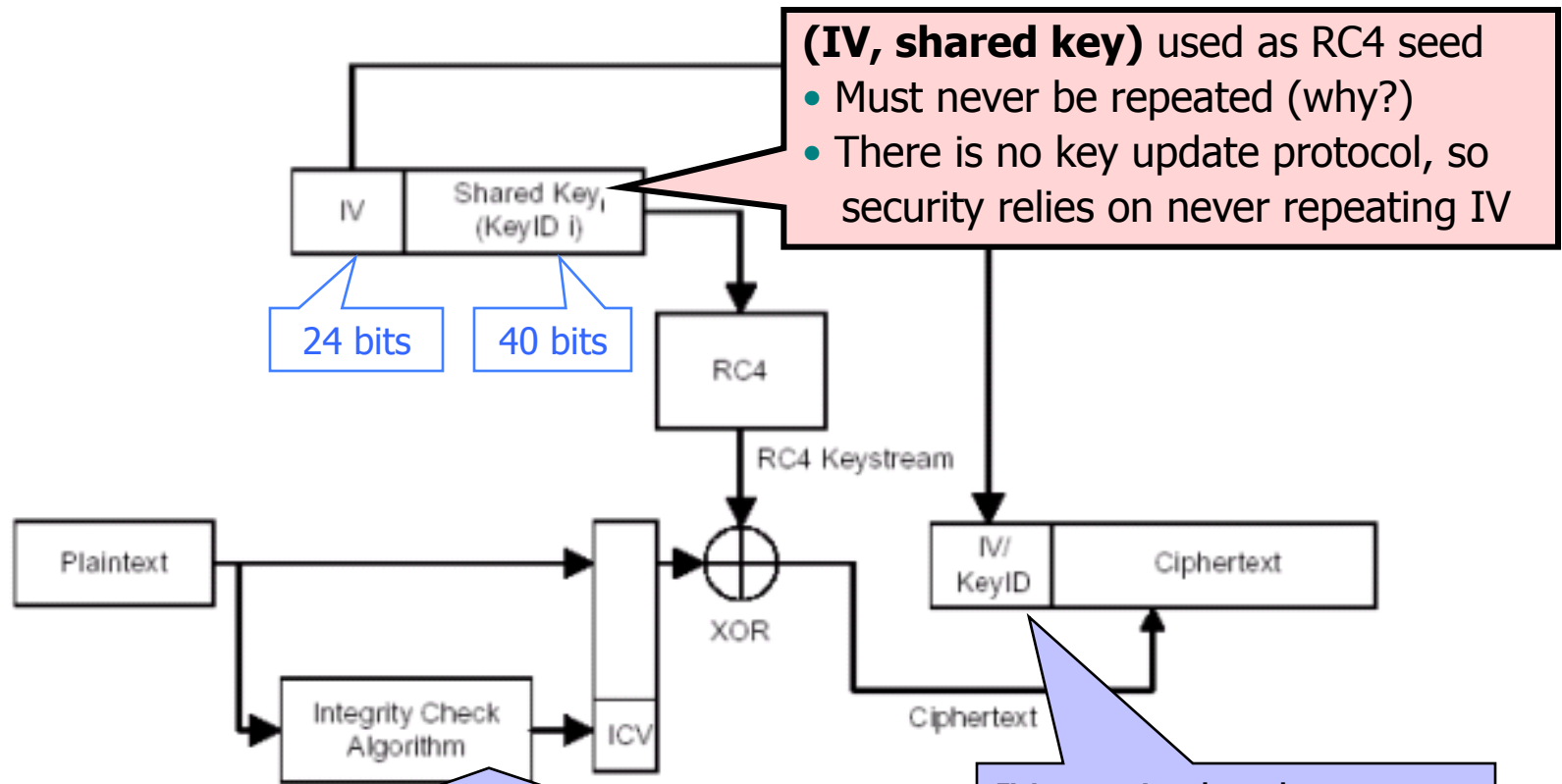
- ◆ Special-purpose protocol for 802.11b
- ◆ Goals: confidentiality, integrity, authentication
 - Intended to make wireless as secure as wired network
- ◆ Assumes that a secret key is shared between access point and client
- ◆ Uses RC4 stream cipher seeded with 24-bit initialization vector and 40-bit key
 - Terrible design choice for wireless environment

Shared-Key Authentication

Prior to communicating data, access point may require client to authenticate



How WEP Works



CRC-32 checksum is linear in \oplus :
if attacker flips some plaintext bits, he knows which bits of CRC to flip to produce the same checksum

no integrity!

IV sent in the clear
Worse: changing IV with each packet is optional!

RC4 Is a Bad Choice for Wireless

- ◆ Stream ciphers require sender and receiver to be at the same place in the keystream
 - Not suitable when packet losses are common
- ◆ WEP solution: a separate keystream for each packet (requires a separate seed for each packet)
 - Can decrypt a packet even if a previous packet was lost
- ◆ But there aren't enough possible seeds!
 - RC4 seed = 24-bit initialization vector + fixed key
 - Assuming 1500-byte packets at 11 Mbps,
 2^{24} possible IVs will be exhausted in about 5 hours
- ◆ Seed reuse is **deadly** for stream ciphers

Recovering the Keystream

- ◆ Get access point to encrypt a known plaintext
 - Send spam, access point will encrypt and forward it
 - Get victim to send an email with known content
- ◆ With known plaintext, easy to recover keystream
 - $C \oplus M = (M \oplus \text{RC4}(\text{IV}, \text{key})) \oplus M = \text{RC4}(\text{IV}, \text{key})$
- ◆ Even without knowing the plaintext, can exploit plaintext regularities to recover partial keystream
 - Plaintexts are not random: for example, IP packet structure is very regular
- ◆ Not a problem if the keystream is not re-used

Keystream Will Be Re-Used

- ◆ In WEP, repeated IV means repeated keystream
- ◆ Busy network will repeat IVs often
 - Many cards reset IV to 0 when re-booted, then increment by 1 \Rightarrow expect re-use of low-value IVs
 - If IVs are chosen randomly, expect repetition in $O(2^{12})$ due to birthday paradox
- ◆ Recover keystream for each IV, store in a table
 - $(\text{KnownM} \oplus \text{RC4}(\text{IV}, \text{key})) \oplus \text{KnownM} = \text{RC4}(\text{IV}, \text{key})$
- ◆ Wait for IV to repeat, decrypt, enjoy plaintext
 - $(M' \oplus \text{RC4}(\text{IV}, \text{key})) \oplus \text{RC4}(\text{IV}, \text{key}) = M'$

It Gets Worse

- ◆ Misuse of RC4 in WEP is a design flaw with no fix
 - Longer keys do not help!
 - The problem is re-use of IVs, their size is fixed (24 bits)
 - Attacks are passive and very difficult to detect
- ◆ Perfect target for the Fluhrer et al. attack on RC4
 - Attack requires known IVs of a special form
 - WEP sends IVs in plaintext
 - Generating IVs as counters or random numbers will produce enough “special” IVs in a matter of hours
- ◆ This results in **key recovery** (not just keystream)
 - Can decrypt even ciphertexts whose IV is unique

Fixing the Problem

- ◆ Extensible Authentication Protocol (EAP)
 - Developers can choose their own authentication method
 - Passwords (Cisco EAP-LEAP), public-key certificates (Microsoft EAP-TLS), passwords OR certificates (PEAP), etc.
- ◆ 802.11i standard fixes 802.11b problems
 - Patch (TKIP): still RC4, but encrypts IVs and establishes new shared keys for every 10 KBytes transmitted
 - Use same network card, only upgrade firmware
 - Deprecated by the Wi-Fi alliance
 - Long-term: AES in CCMP mode, 128-bit keys, 48-bit IVs
 - Block cipher in a stream cipher-like mode

Hacking MIFARE Chips

- ◆ Multi-year project on evaluating security of MIFARE cards at Radboud University in Holland
 - <http://www.ru.nl/ds/research/rfid/>
- ◆ MIFARE = a case study in how not to design cryptographic authentication systems
- ◆ The following slides are from Peter Van Rossum



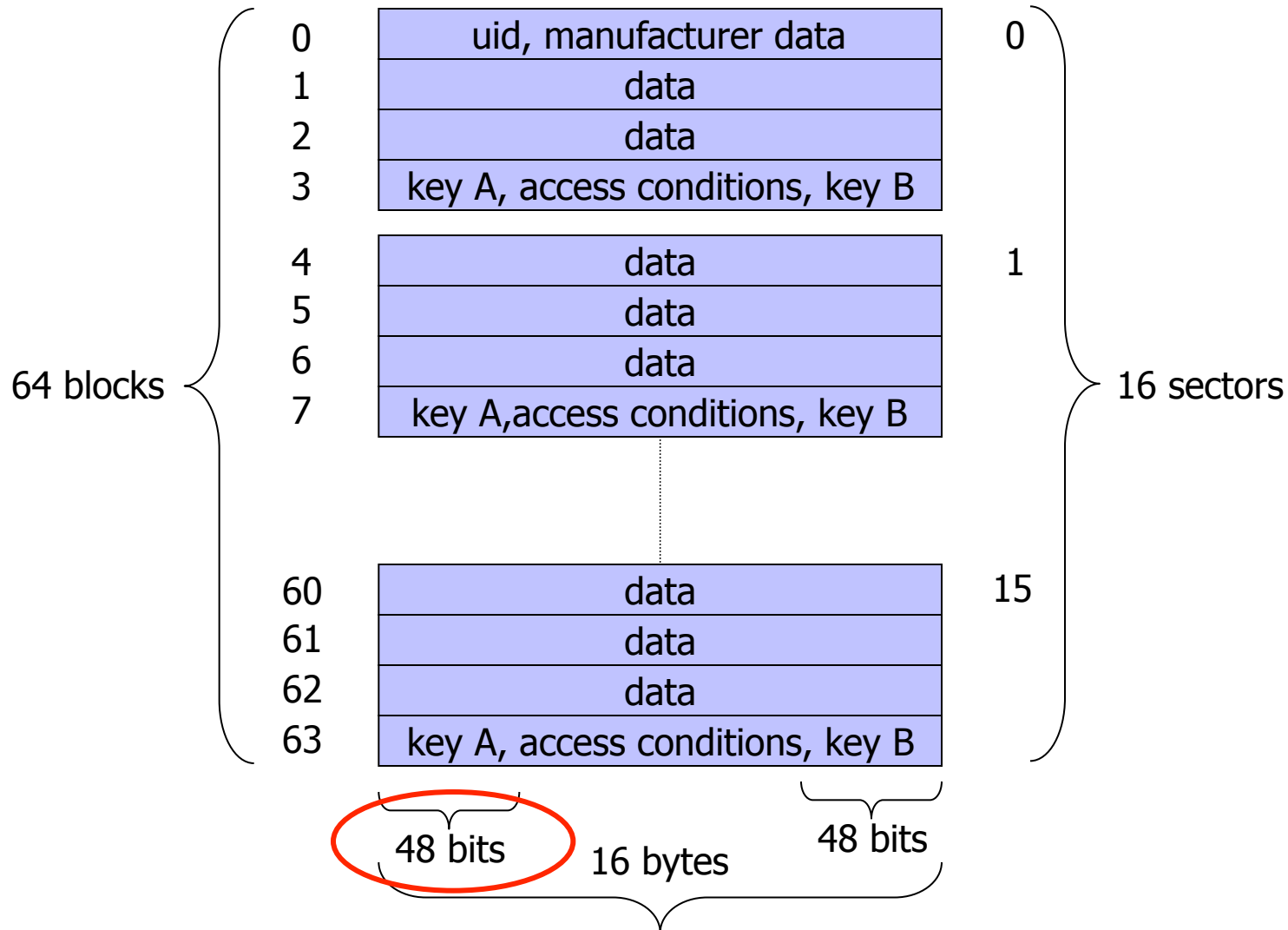
MIFARE Chips

- ◆ Series of chips used in contactless smart cards
 - Developed by NXP Semiconductors in the Netherlands
- ◆ Very common in transport payment cards

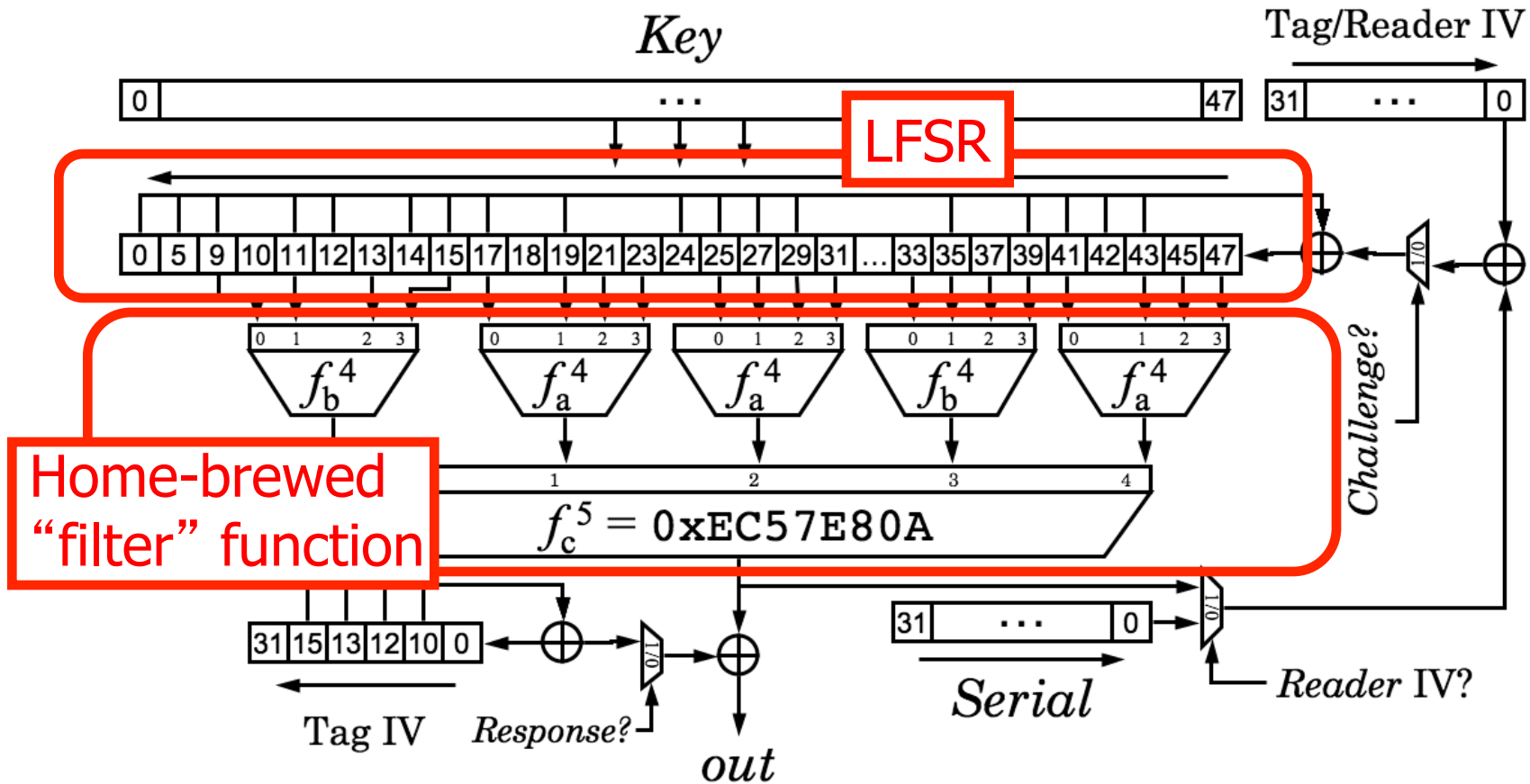


- ◆ MIFARE Classic: 80% of the market
 - Over 1 billion sold, over 200 million in use

Memory Structure of the Card



Crypto1 Cipher

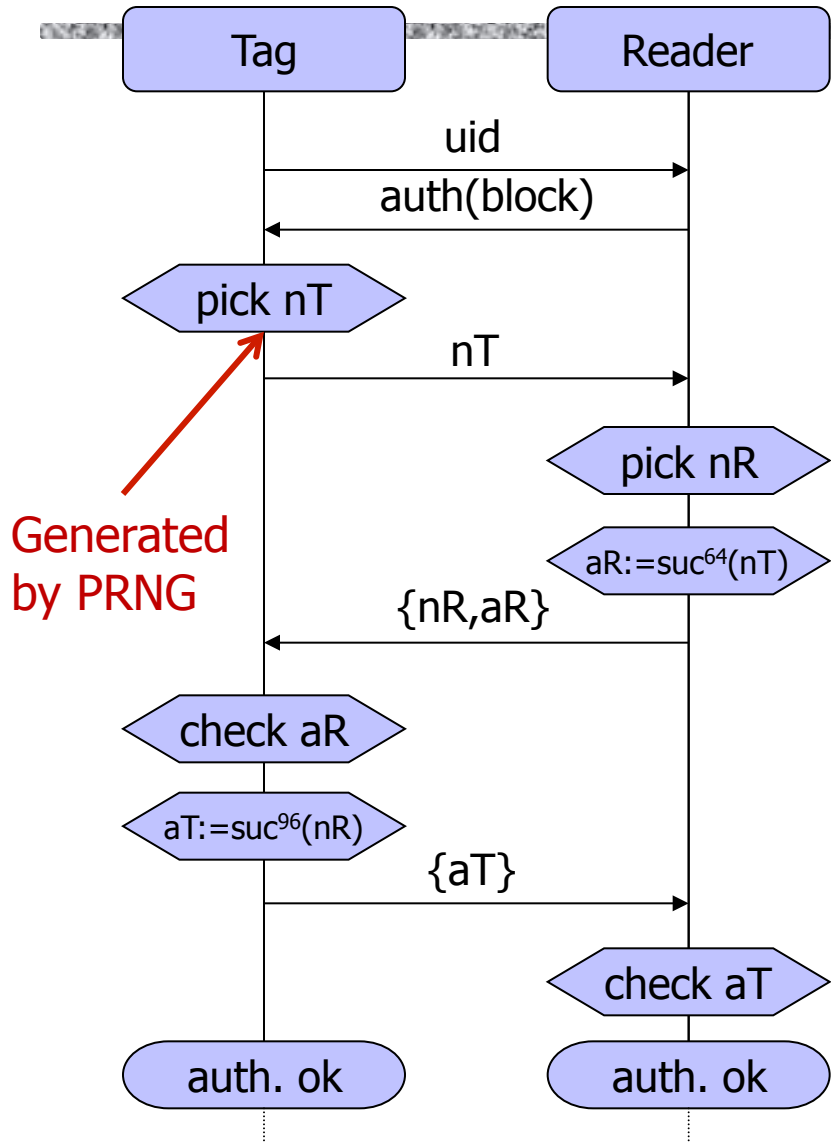


$$f_a^4 = 0x9E98 = (a+b)(c+1)(a+d)+(b+1)c+a$$

$$f_b^4 = 0xB48E = (a+c)(a+b+d)+(a+b)cd+b$$

Tag IV \oplus Serial is loaded first, then Reader IV \oplus NFSR

Challenge-Response in CRYPTO1



LFSR stream:

Initial state of the LFSR is the key

$$a_i := k_i \quad i \in [0,47]$$

Shift $nT + uid$ into the LFSR

$$a_{i+48} := L(a_i, \dots, a_{i+47}) + nT_i + uid_i \quad i \in [0,31]$$

Shift nR into the LFSR

$$a_{i+48} := L(a_i, \dots, a_{i+47}) + nR_{i-32} \quad i \in [32,63]$$

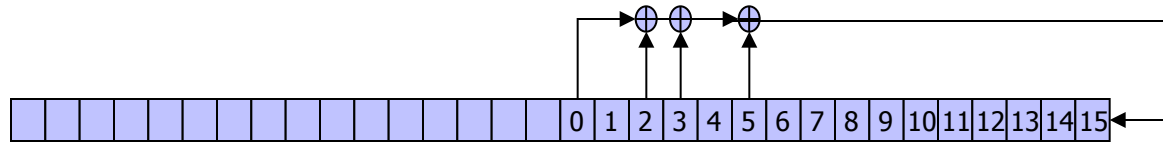
After authentication, LFSR keeps shifting

$$a_{i+48} := L(a_i, \dots, a_{i+47}) \quad i \in [64, \infty)$$

Keystream:

$$b_i := f(a_{i+9}, a_{i+11}, \dots, a_{i+47}) \quad i \in [32, \infty) \quad \text{slide 29}$$

PRNG in CRYPTO1



- Linear feedback shift register
- 16-bit internal state
- Period $2^{16} - 1 = 65535$

Feedback:

$$L_{16}(x_0, x_1, \dots, x_{15}) := x_0 + x_2 + x_3 + x_5$$

Successor:

$$\text{SUC}(x_0, x_1, \dots, x_{31}) := (x_1, x_2, \dots, x_{30}, L_{16}(x_{16}, x_{17}, \dots, x_{31}))$$

Replay Attack

[Gans, Hoepman, Garcia]

- ◆ Good challenge-response authentication requires some form of “freshness” in each session
 - For example, timestamp or strong (pseudo)randomness
- ◆ MIFARE Classic: no clock + weak randomness
 - “Random” challenges repeat a few times per hour
- ◆ Eavesdrop and record communication session
- ◆ When challenge repeats, send known plaintext, extract keystream, use it to decrypt recorded communication that used the same challenge

Extracting the Key from Reader

1. Acquire keystream

- Observe authentication → keystream
- 1 to 3 authentication sessions – takes microseconds

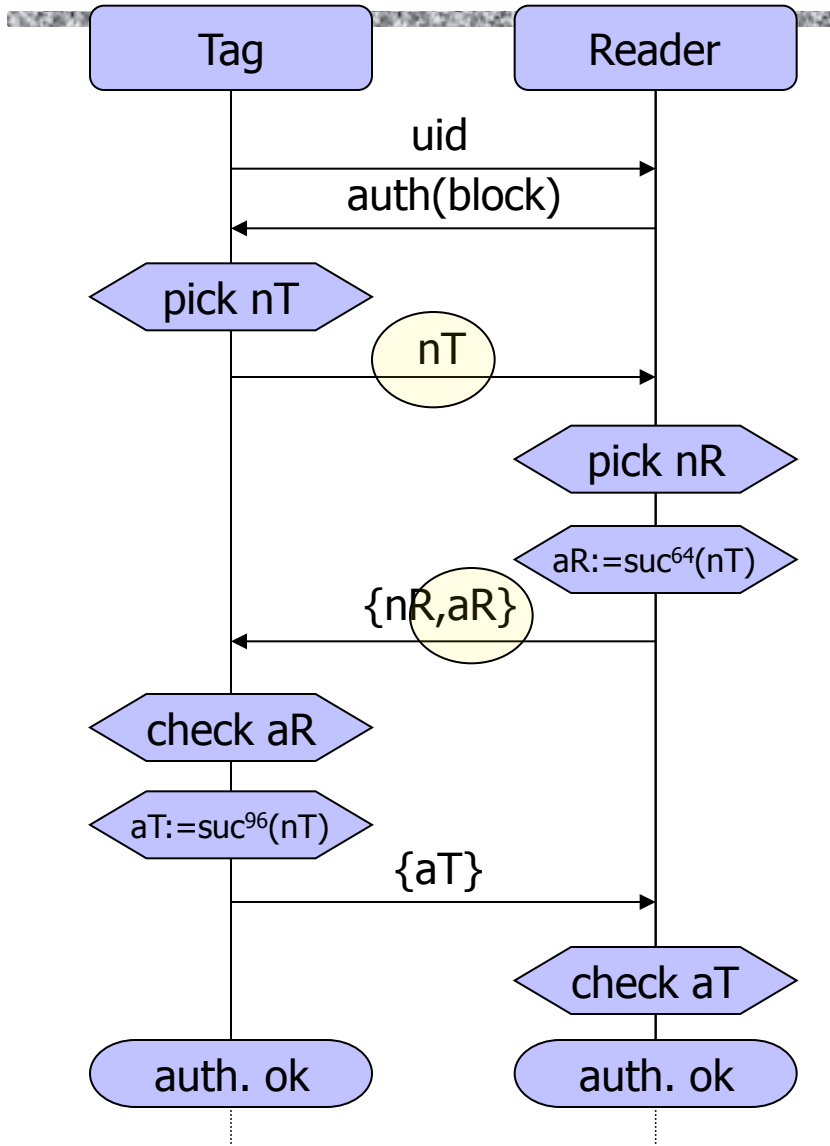
2. Invert the filter function

- Keystream → internal state of LFSR
- Approx. 2^{26} operations – takes seconds

3. Roll back (“unshift”) the LFSR

- Internal state of LFSR at any time → seed (= key)
- Problem: bad PRNG design... cryptographically secure PRNG should not allow rollback and recovery of the seed even if state is compromised

Acquiring Keystream



Intercepted communication:

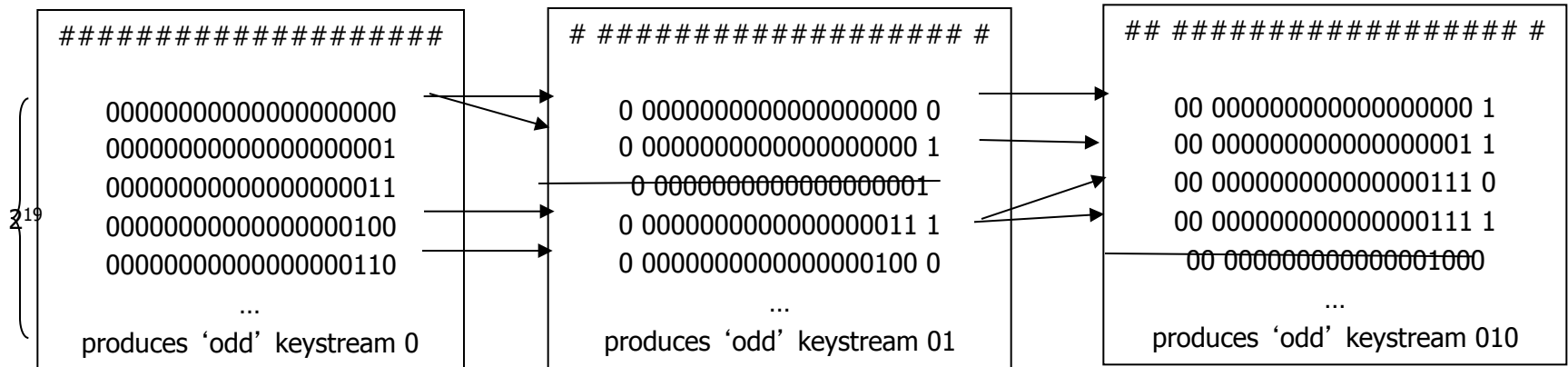
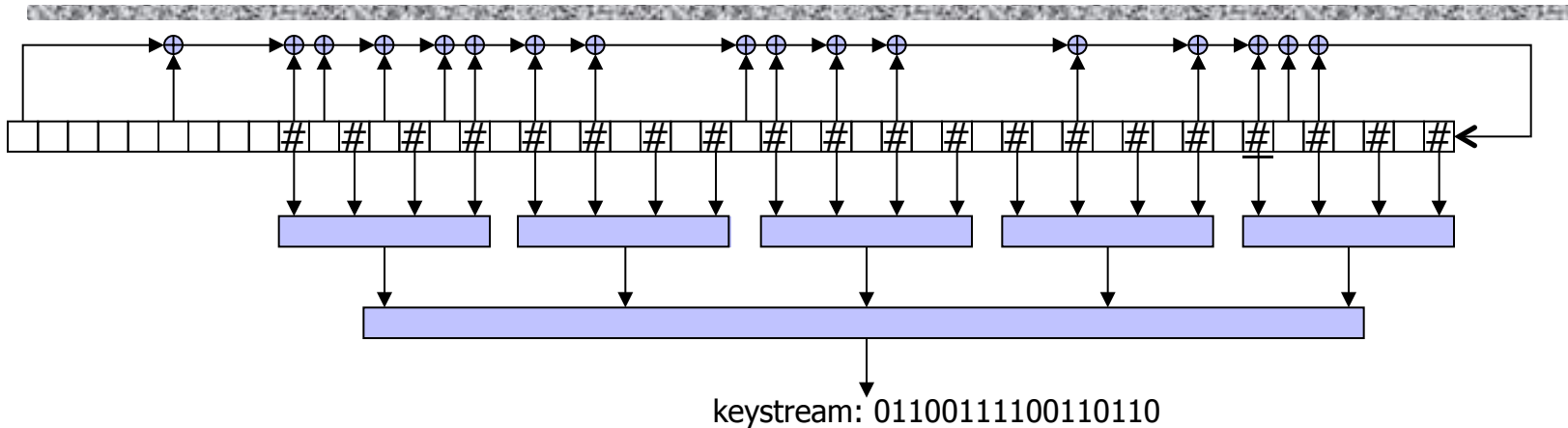
- nT , $\{aR\}$, $\{aT\}$ visible to attacker
- $\{aR\} = \text{suc}^{64}(nT)$, $\{aT\} = \text{suc}^{96}(nT)$
- 64 keystream bits

OR

Access to reader only:

- nT under attacker control
- $\{aR\} = \text{suc}^{64}(nT)$ visible to attacker
- 32 keystream bits

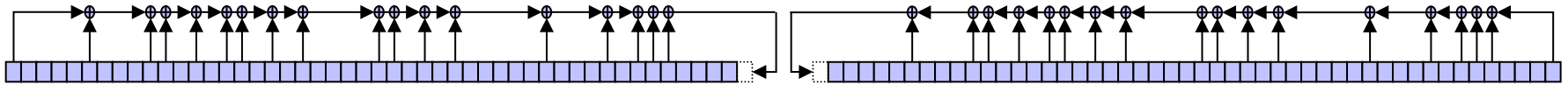
Inverting the Filter Function



Filter function only depends only on 20 odd bits of input → easily inverted

- Compute 'odd' bits of LFSR using table and deduce 'even' bits (linear relation) OR
- Compute 'odd' and 'even' bits of LFSR using tables separately and combine tables

Rolling Back the LFSR



Feedback:

$$L(x_0, x_1, \dots, x_{47}) := x_0 + x_5 + x_9 + x_{10} + x_{12} + x_{14} \\ + x_{15} + x_{17} + x_{19} + x_{24} + x_{25} + x_{27} + x_{29} + x_{35} + x_{39} \\ + x_{41} + x_{43}$$

LFSR stream:

Initial state of the LFSR is the key

$$a_i := k_i \quad i \in [0, 47]$$

Shift $nT + uid$ into the LFSR

$$a_{i+48} := L(a_i, \dots, a_{i+47}) + nT_i + uid_i \quad i \in [0, 31]$$

Shift nR into the LFSR

$$a_{i+48} := L(a_i, \dots, a_{i+47}) + nR_{i-32} \quad i \in [32, 63]$$

After authentication, LFSR keeps shifting

$$a_{i+48} := L(a_i, \dots, a_{i+47}) \quad i \in [64, \infty)$$

Keystream:

$$b_i := f(a_{i+9}, a_{i+11}, \dots, a_{i+47}) \quad i \in \mathbb{N}$$

Inverting feedback:

$$R(x_1, \dots, x_{47}, x_{48}) := x_5 + x_9 + x_{10} + x_{12} + x_{14} \\ + x_{15} + x_{17} + x_{19} + x_{24} + x_{25} + x_{27} + x_{29} + x_{35} + x_{39} \\ + x_{41} + x_{43} + x_{48}$$

$$R(x_1, \dots, x_{47}, L(x_0, x_1, \dots, x_{47})) = x_0$$

Inverting LFSR stream:

Unshift LFSR until end of authentication

$$a_i = R(a_{i+1}, \dots, a_{i+48}) \quad i \in [64, \infty)$$

Unshift nR from the LFSR

$$a_i = R(a_{i+1}, \dots, a_{i+48}) + nR_{i-32} \quad i \in [32, 63]$$

$$= R(a_{i+1}, \dots, a_{i+48}) + \{nR\}_{i-32} + b_i$$

$$= R(a_{i+1}, \dots, a_{i+48}) + \{nR\}_{i-32} + f(a_{i+9}, \dots, a_{i+47})$$

Unshift $nT + uid$ from the LFSR

$$a_i = R(a_{i+1}, \dots, a_{i+48}) + nT_i + uid_i \quad i \in [0, 31]$$

Key is the initial state of the LFSR

$$k_i = a_i \quad i \in [0, 47]$$

Summary: Weaknesses of CRYPTO1

- ◆ Stream cipher with 48-bit internal state
 - Enables brute-force attack
- ◆ Weak 16-bit random number generator
 - Enables chosen-plaintext attack and replay attack
- ◆ Keystream based on simple LFSR structure + weak “one-way” filter function
 - Invert filter function → obtain state of LFSR
 - Roll back LFSR → recover the key
 - 64-bit keystream → recover unique key
 - 32-bit keystream → 216 candidate keys

Extracting the Key (Card Only)

- ◆ Parity bit of plaintext is encrypted with the same bit of the keystream as the next bit of plaintext
 - “One-time” pad is used twice
- ◆ If parity bit is wrong, encrypted error message is sent before authentication
 - Opens the door to card-only guessing attacks (chosen-plaintext, chosen-ciphertext) – why?
 - Wireless-only attack
- ◆ Recover secret key from the card in seconds
 - Result: full cloning of the card