CT437
COMPUTER SECURITY AND FORENSIC COMPUTING

STREAM CIPHERS

Dr. Michael Schukat

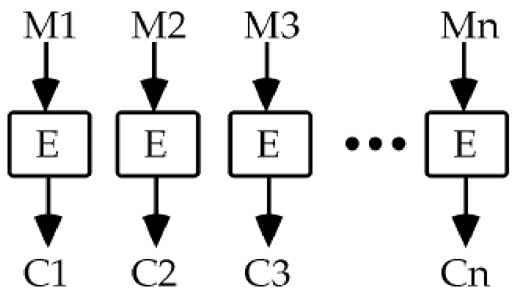
Ollscoil na Gaillimhe
University of Galway

Lecture Overview

- This slide decks covers the following topics:
  - Stream Ciphers and their implementation in
    - LFSR
    - NLFSR
    - RC4
  - Pseudorandom number generation principles

Recap: Block Ciphers versus Stream Ciphers

- In a block cipher the data (e.g. text, video, or a network packet) to be encrypted is broken into blocks M1, M2, etc. of K bits length, each of which is then encrypted
- The encryption process is like a substitution on very big characters – 64 bits or more



encoding

decoding

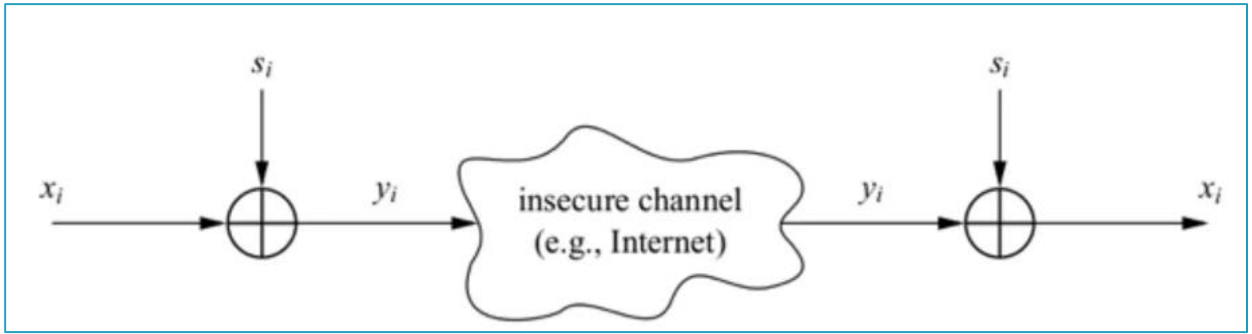
- In contrast, a **stream cipher** is a symmetric key cipher where plaintext digits are combined with a pseudorandom cipher digit stream (the **keystream**)
- Normally,
  - stream ciphers only process one bit or one byte at a time
  - the combining operation is an exclusive-or (XOR)

Stream Ciphers

- Stream ciphers typically provide a (pseudo) random stream key generator that produces a pseudo-random digit sequence s_i (i = 1, 2, ...)
- This stream is XORed digit-by-digit with the plaintext x:

$$y_i = x_i \text{ XOR } s_i$$

- The plaintext stream can be recovered by reapplying the XOR operation
- In modern stream ciphers, a digit is one bit (or one byte → later)
- A random stream key completely destroys any statistically properties in the plaintext message
  - For a perfectly random keystream s_i, each y_i has a 50% chance of being 0 or 1
- But how can a pseudo-random sequence s_i be generated?



x_i	s_i	y_i
0	0	0
0	1	1
1	0	1
1	1	0

Stream Cipher Performance

- Since an XOR operation of a single bit or byte can be done in a single CPU cycle,
  - the code size and complexity of a stream cipher mainly depends on the code size and complexity of the random number generator
  - the speed of a stream cipher mainly depends on the speed of the random number generator
- For comparison (based on some Intel Pentium architecture):

Cipher	Key length	Mbit/s
DES	56	36.95
3DES	112	13.32
AES	128	51.19
RC4 (stream cipher)	(choosable)	211.34

- Size and speed make stream ciphers very suitable for resource constrained devices (e.g., mobile phones, IoT devices)

One-Time Pad

- The OTP is an encryption requires the use of a single-use pre-shared key that is equal to the size of the message being encrypted
- For the resulting ciphertext to be impossible to decrypt, the key must…
  - be at least as long as the plaintext (think of Vigenère and its weakness)
  - be
    - random (uniformly distributed in the set of all possible keys and independent of the plaintext)
    - entirely sampled from a non-algorithmic, chaotic source such as a hardware random number generator
    - pattern-less
  - never be reused in whole or in part (Coincidence counting -> next slide)
  - be kept completely secret by the communicating parties

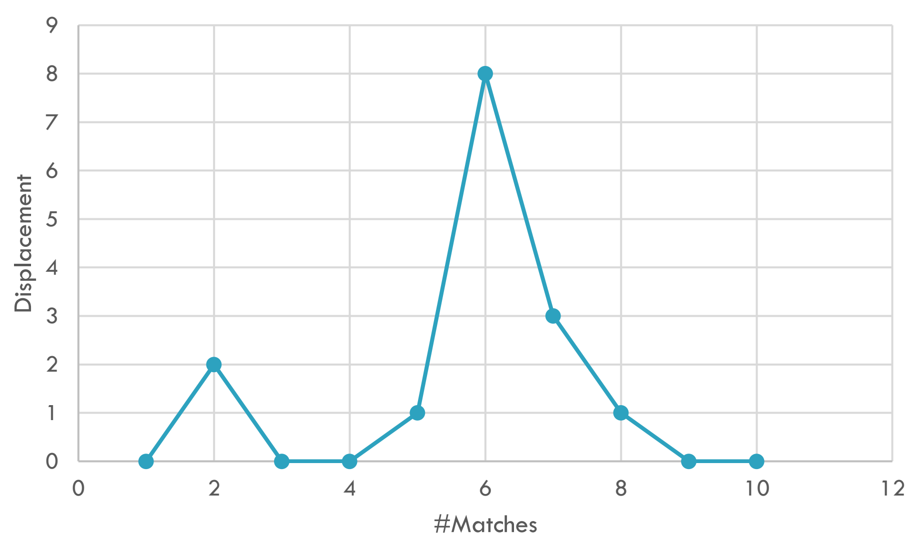
- OTPs are not practical for practical reasons, therefore pseudo-random generators (PRG) are used
- PRGs are often based on Linear Feedback Shift Registers (LFSRs)

Example Coincidence Counting

- Coincidence counting allows predicting the length of the key of a stream cipher, by comparing the ciphertext against itself with different offsets
- Assume ciphertext CXEKCWCOZKUCAYZEKW that has been encoded using a stream cipher with an unknown key
- Count the number of identical characters (matches) using different displacements of ciphertext:
  - Displacement = 1
    ```
    CXEKCWCOZKUCAYZEKW
     CXEKCWCOZKUCAYZEKW
    ```
    Matches: 0
  - Displacement = 2
    ```
    CXEKCWCOZKUCAYZEKW
      CXEKCWCOZKUCAYZEKW
    ```
    Matches: 1
  - Displacement = 3
    ```
    CXEKCWCOZKUCAYZEKW
       CXEKCWCOZKUCAYZEKW
    ```
    Matches: 0
  - …

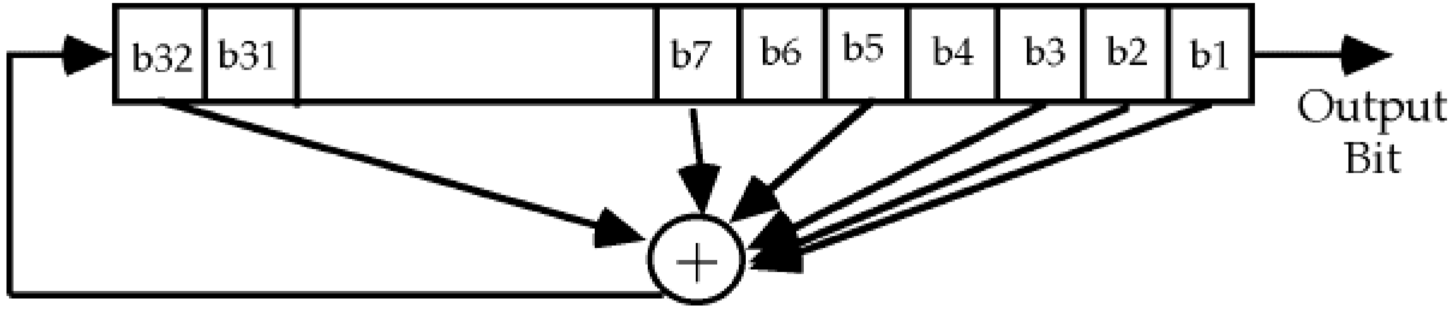
Example Coincidence Counting

- If you line up the ciphertext with itself displaced by k (= key length) characters, then you get a match in the ciphertext (offset by k places) if there is a match in the plaintext (offset by k places)
  - With the non-uniformity of the frequency distribution of English letters there's about a 6% chance that those two positions have the same letter (the index of coincidence)
- In contrast, when you line up the ciphertext using a different displacement, the index of coincidence is much smaller, i.e., 1/256, if ciphertexts are bytes
- By counting the displacement over a long ciphertext stream, k can be determined



Linear Feedback Shift Registers (LFSR)

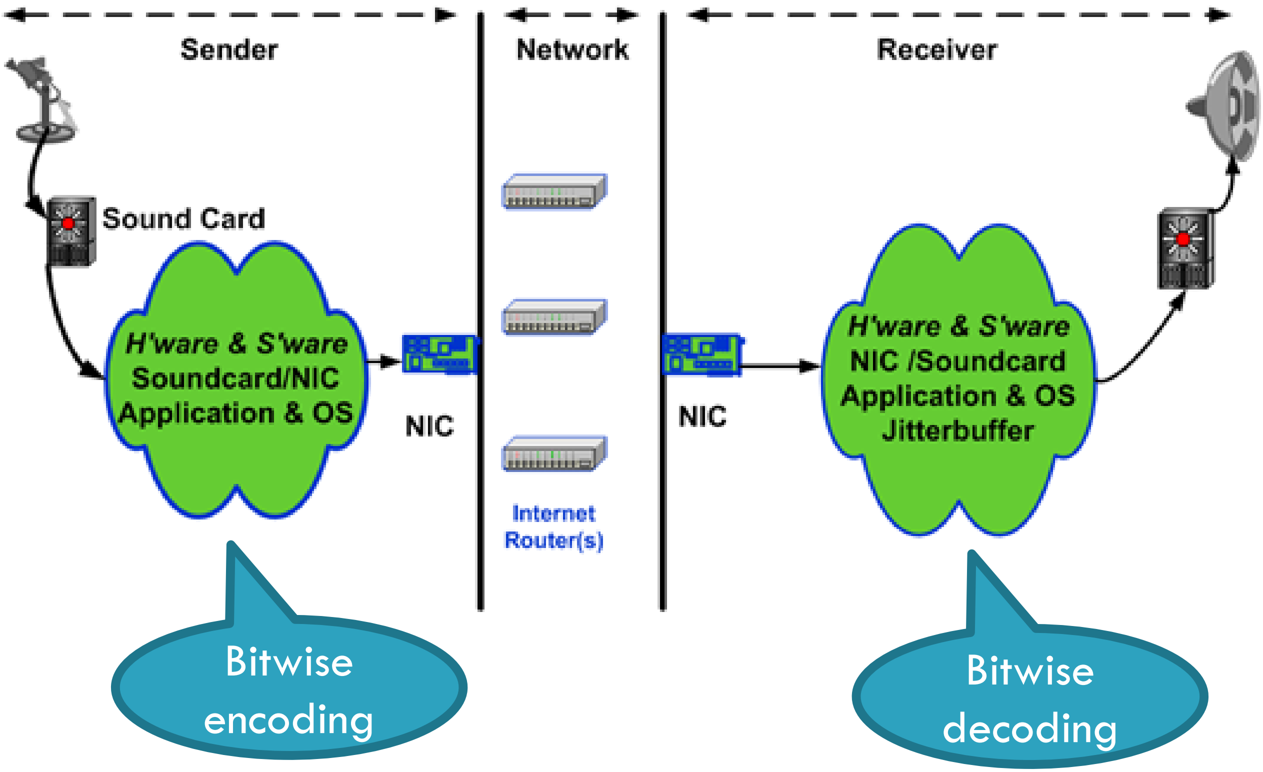
- A LFSR consists of a binary shift register of some length along with a linear feedback function (LFF) that operates on some of those bits
  - The most commonly used LFF is the XOR operation
- To get started the register is preset with a secret initialisation vector
- Each time a bit is needed,
  - a new bit is formed from the linear feedback function
  - all bits are shifted by one position (shifted right in the example below) with the new bit being shifted in
- The bit shifted out is used as the (pseudo-random) output of the LFSR
- A well-designed n-bit LFSR generates a pseudo-random sequence whose length correlates to n



Example for an 8-Bit LFSR

- Initialisation vector: 10100110 ($B_7 \cdots B_0$)
- Feedback Function: B_7 XOR B_4 XOR B_1
- Right shift after each cycle (B_0 shifted out)
- Iteration 0: 10100110
- Iteration 1: 01010011 >> 0
- Iteration 2: 00101001 >> 1
- Iteration 3: 00010100 >> 1
- Iteration 4: 10001010 >> 0
- … …

Example VoIP Encoding using a Stream Cipher



Stream Ciphers in Practice

- In practice, one key is used to encrypt many messages
  - Example: Wireless communication
  - Solution: Use Initial vectors (IV)
  - $E_{key}[M] = [IV, M \oplus PRNG(key \mid\mid IV)]$
    - IV is sent in clear to receiver
    - IV needs integrity protection, but not confidentiality protection
    - IV ensures that key streams do not repeat, but does not increase cost of brute-force attacks
    - Without key, knowing IV still cannot decrypt
  - Need to ensure that IV never repeats! How?

Example for a 16-bit LFSR written in C


```
#include <stdint.h>
#include <stdio.h>
int main(void) {
    uint16_t start_state = 0xACE1u;  /* Any non-zero start state will work. */
    uint16_t lfsr = start_state;
    uint16_t bit, input, period = 0;
    printf("Enter LFSR IV as integer: "); scanf("%d", &input);
    if (input > 0) {
       start_state = input;
       lfsr = start_state;
    }
    do
    {   /* LFF: B15 XOR B13 XOR B12 XOR B10 */
        bit = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5)) & 1u;
        lfsr = (lfsr >> 1) | (bit << 15);
        printf("%d", bit);
        ++period;
    } while (lfsr != start_state);
    printf("\nPeriod of output sequence: %d \n", period);
    return 0;
}
```

What is the Maximum Sequence Length of a single LFSR?

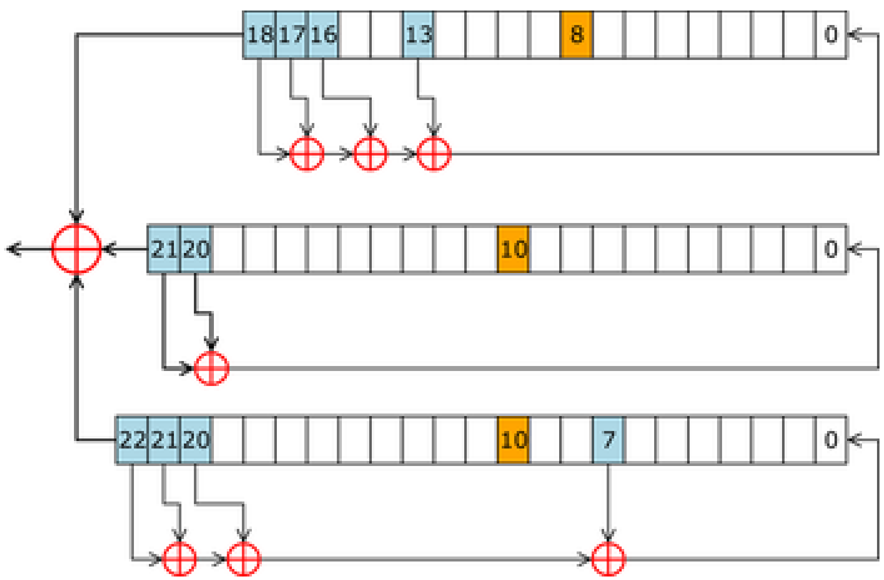
- Consider a single n-bit LFSR with some feedback function
- Each bit that is shifted out is intrinsically linked to the content of the LFSR
- Each shift operation maps the register content to another (different) pattern, as seen in the example, resulting in another bit shifted out
- An n-bit LFSR allows for 2^n different register content variations, with each variation pushing out a 0 or a 1
- Therefore, the longest cycle of non-repeating patterns is $2^n – 1$ iterations, with 2^n the maximum length of the sequence
  - Think of a 1-bit LFSR (n = 1):
    - There are 2 different LFSR contents ("0" or "1") possible
    - The longest possible patterns are "10" or "01"; both have a length of 2^n
    - It just takes one iteration (2^{n-1}) to reach all possible register contents (1 → 0 or 0 → 1)
- However,
  - poorly designed LFSR may result in cycles that are shorter
  - the Index of Coincidence problem also applies to LFSR (and in fact to all stream ciphers)

The Combined LFSR

- A combined LFSR uses multiple LFSR in parallel, and combines their respective outputs to generate a key stream
- They work well on resource-constrained devices too
- Example: A5/1, which was used for GSM voice communication:
  - The Global System for Mobile Communications (GSM) was a mobile phone standard back in the 1990s
  - In GSM, digitised phone conversations are sent as sequences of frames
  - One frame is sent every 4.6 milliseconds and is 228 bits in length
    - Voice samples are collected / digitised over 4.6 milliseconds and send in a block
  - A5/1 is a combined LFSR-based algorithm that is used to produce 228 bits of key stream which is XORed with the frame
  - It is initialised using a 64-bit key

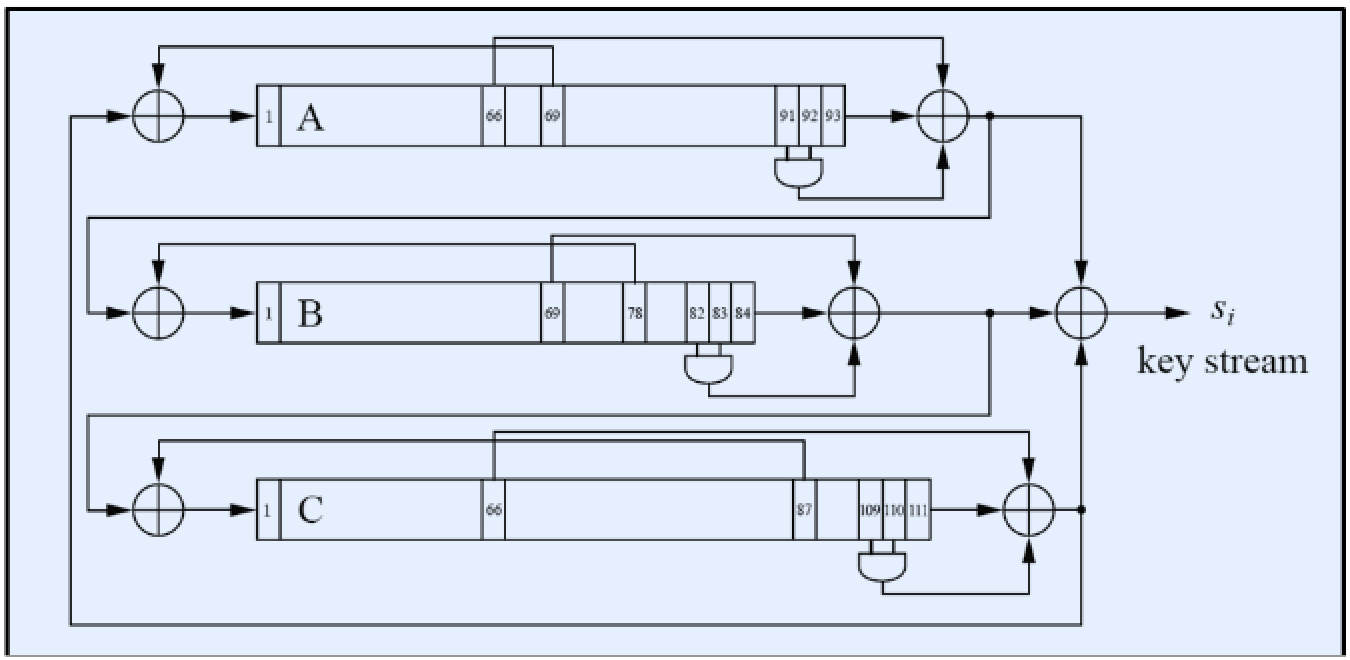
Example A5/1

- 3 independent LFSRs:
  - LFSR 1
    - 19 bits
    - LFF: B18 XOR B17 XOR B16 XOR B13
  - LFSR 2:
    - 22 bits
    - LFF: B21 XOR B20
  - LFSR 3:
    - 23 bits
    - LFF: B22 XOR B21 XOR B20 XOR B7
- The output bit is the XORed output of all 3 LFSRs
- A LFSR is only shifted to the left, if their clocking bit (B8, B10, and B10 respectively) matches the output bit;
  otherwise, there is no shift, and the same output bit value is used again in the next cycle



Non-Linear Feedback Shift Registers (NLFSR)

- NLFSR contain AND gates as well as XOR gates in their feedback function
- Example Trivium: A, B and C are three shift registers with bit lengths of 93, 84 and 111 bits respectively



Example for a 16-bit NLFSR in C


```
#include <stdint.h>
#include <stdio.h>
int main(void)
{
    uint16_t start_state = 0xACE1u;  /* Any non-zero start state will work. */
    uint16_t lfsr = start_state;
    uint16_t bit, period = 0;
    do
    {   /* FBF: B15 XOR B13 XOR B12 XOR B10 XOR (B2 and B1)*/
        bit = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5) ^ ((lfsr >> 13) & (lfsr >> 14))) & 1u;
        lfsr = (lfsr >> 1) | (bit << 15);
        printf("%d", bit)
        ++period;
    } while (lfsr != start_state);
    printf("\nPeriod of output sequence: %d \n", period);
    return 0;
}
```

Pseudo-Random Number generation: RC4

- Instead of single bits, a generator algorithm can also produce one byte (or one word) at a time
- RC4 is an example for such an algorithm, it returns one pseudorandom byte at a time
- It was designed by Ron Rivest of RSA Security in 1987
- RC4 was initially a trade secret, but in 1994 a description of it was anonymously posted on the Internet
- RC4 consists of a
  - key-scheduling algorithm (KSA) and a
  - pseudo-random generation algorithm (PRGA)

RC4: The Key-Scheduling Algorithm (KSA)

- The KSA requires a key (stored in `key[]`) of length `keylength`
  - `keylength` is somewhere between 1 and 256
- Using the keyword, a 256-byte long permutation vector `S[]` is generated:

```
for i from 0 to 255
    S[i] := i;
j := 0;
for i from 0 to 255
    j := (j + S[i] + key[i mod keylength])
          mod 256;
    swap(S[i],S[j]);
```

RC4: The Pseudo-Random Generation Algorithm (PRGA)

- PRGA returns one byte at a time:

```
i := 0;
j := 0;
while GeneratingOutput:
    i := (i + 1) mod 256;
    j := (j + S[i]) mod 256;
    swap(S[i],S[j]);
    output S[(S[i] + S[j]) mod 256];
```

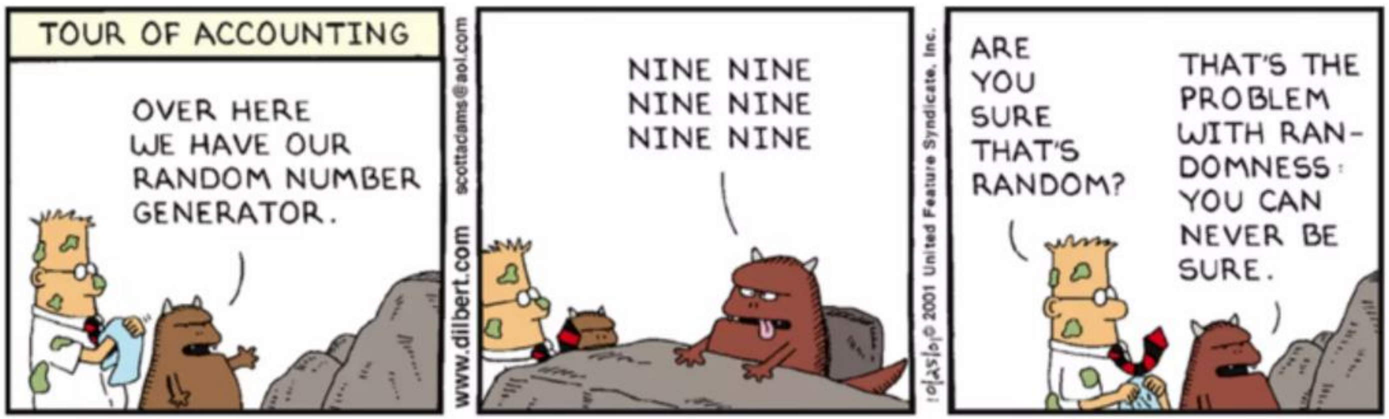
Security of RC4

- Obviously not an LFSR-based design, but a more general pseudo-random number generator design
- Can also be efficiently implemented in software
  - Very compact algorithm
- However, it is not deemed safe anymore!

3 Security
- 3.1 Roos's biases and key reconstruction from permutation
- 3.2 Biased outputs of the RC4
- 3.3 Fluhrer, Mantin and Shamir attack
- 3.4 Klein's attack
- 3.5 Combinatorial problem
- 3.6 Royal Holloway attack
- 3.7 Bar-mitzvah attack
- 3.8 NOMORE attack

Background: Pseudorandom Number Generators

- Cryptographically strong pseudorandom number generation is essential!



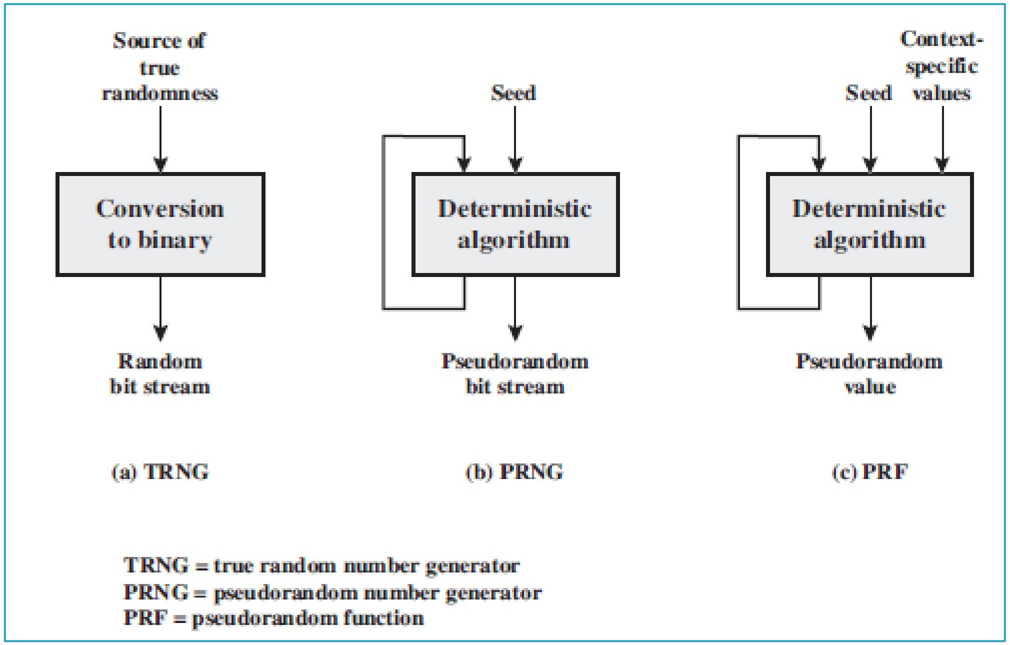
- Pseudorandom number generators (PRNG) are used in a variety of cryptographic and security applications, including
  - Stream cipher encryption → 802.11 WEP
  - Encryption keys (both for symmetric and public key algorithms)

Obvious Requirements for Random Number Generators

- Assume we toss a fair coin or throw a fair dice multiple times. We expect the following from the resulting sequence:
- Randomness, i.e. uniform distribution
  - The distribution of values in the sequence (e.g. “head or tail”) should be uniform; that is, the frequency of occurrence of possible outputs should be approximately equal
- Unpredictability, i.e. independence
  - Successive members of the sequence are unpredictable; no subsequence in the sequence can be inferred from the others

Types of Random Generators

- A TRNG takes as input a source that is effectively random
  - The source is often referred to as an **entropy source**
  - The entropy source is drawn from the physical environment of the computer, e.g. a combination of keystroke timing patterns, CPU temperature changes and mouse movements



- A PRNG uses just a seed (e.g. LFSR)
- A PRF often also takes in a context-specific value, e.g.
  - A secure end-to-end communication via TCP/IP may take in the endpoints' IP addresses
- However, PRNG and PRF are based on deterministic algorithms, therefore the "P"

Formal Requirements for Pseudorandom Generators

- Randomness
  The generated bit stream must "appear" random even though it is deterministic
  This can be validated by applying a sequence of tests to the generator, which determine (among others) the following characteristics:
  - Uniformity: At any point in the generation of a sequence of random or pseudorandom bits, the occurrence of a zero or one is equally likely; The expected number of zeros (or ones) is n/2, with n being the sequence length
  - Scalability: Any test applicable to a sequence can also be applied to sub-sequences extracted at random; if a sequence is random, then any such extracted subsequence should also be random
  - Consistency: The behavior of a generator must be consistent across many starting values (seeds); it is inadequate to test a PRNG based on the output from a single seed

Formal Requirements for Pseudorandom Generators

- Unpredictability
  A stream of pseudorandom numbers should exhibit two forms of unpredictability
  - Forward unpredictability: If the seed is unknown, the next output bit in the sequence should be unpredictable in spite of any knowledge of previous bits in the sequence
  - Backward unpredictability: It should not be feasible to determine the seed from knowledge of any generated values; no correlation between a seed and any value generated from that seed should be evident; each element of the sequence should appear to be the outcome of an independent random event whose probability is 0.5

NIST SP 800-22

- The National Institute of Standards and Technology (NIST) published the above report, "*A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*"
- It lists 15 separate tests of randomness and unpredictability
- https://github.com/terrillmoore/NIST-Statistical-Test-Suite