Radbound University

Stream ciphers: native stream ciphers and security notion

Cryptography, Autumn 2021

Lecturers: J. Daemen, B. Mennink

September 14, 2021

Institute for Computing and Information Sciences
Radboud University

Outline

Modern stream ciphers

How not to build modern stream ciphers

Some real-world stream ciphers

Modelling attacks on stream ciphers

The Random Oracle

Indistinguishability security notion

Modern stream ciphers

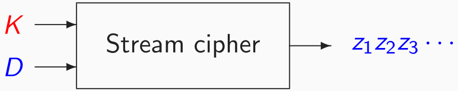
Requirement for re-synchronization

- Stream ciphers discussed up to now
  - input: short cipher key K
  - output: long keystream Z
  - necessary condition: each bit z_t shall be used only once!
- Practical problems:
  - Alice and Bob need to keep cipher (LFSR) states synchronous
  - communication is lost when losing synchronization
- Solution
  - add another input: a diversifier D (AKA initial value)
  - same cipher key can now be used to generate many keystreams
  - for each message encryption use a different value for D

Modern stream ciphers

Modern stream ciphers take a key K and a diversifier D as input

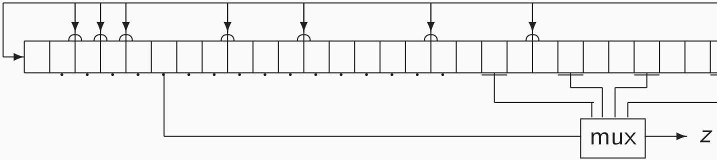
How to use modern stream ciphers



- Message encryption
  - have a system that generates a unique diversifier D per message
  - e.g., date/time, message sequence number, random value, ...
  - encipher message with keystream Z from K and D
- Data streams, e.g., pay TV, telephone, ...
  - split in relatively short, numbered, sub-sequences, e.g., frames
  - keystream to encipher a sub-sequence uses its number as D

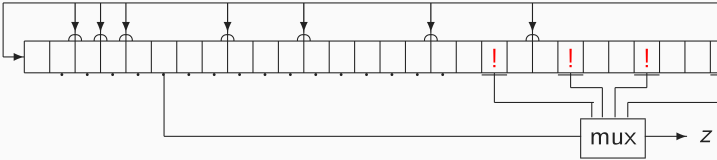
How not to build modern stream ciphers

Multiplexer LFSR supporting diversifier: resync attack



- Real-world stream cipher (early '90s), approach:
  - choose LFSR and mux dimension d to resist best known attacks
  - initialize state with $K+D$
- Adversary may get keystream for multiple, say n, diversifier values:
  - $D^{(0)}$ gives $Z^{(0)}$
  - $D^{(1)}$ gives $Z^{(1)}$
  - $\dots$
  - $D^{(i)}$ gives $Z^{(i)}$
  - $\dots$
- We zoom in on the state $s_t^{(i)}$ and keystream bit $z_t^{(i)}$ at cycle t
- For compactness, we omit the t subscript and write $s^{(i)}$ and $z^{(i)}$

Resync attack (cont'd)



- Some notation:
  - we have $s^{(i)} = M^t(K + D^{(i)}) = M^t K + M^t D^{(i)}$
  - we write K' for $M^t K$ and $E^{(i)}$ for $M^t D^{(i)}$
  - we now have $s^{(i)} = K' + E^{(i)}$:
- Guess-and-determine attack starting at cycle t: $z_t^{(i)}$
  - make hypothesis for 4 bits of K', in mux address positions (!)
  - ...this allows computing corresponding bits of $s^{(i)}$
  - ...and for each, allows appointing $z^{(i)}$ to a bit of $s^{(i)}$
  - each of these n statebits can be converted to K'
    - if wrong hypothesis, huge probability for inconsistency
    - if right hypothesis, known part of the state fills up fast
- Leads to immediate break even if n is quite modest, for any t

Why resync attacks work

- State update function is linear
  - lightweight and convenient to implement in *constant time*
  - analyzable: length of cycle known in advance
  - but difference between states known after init, remains known forever
- Mapping from D to initial state is linear
  - simple and cheap
  - but difference of D's known $\rightarrow$ state difference known
- Output function (multiplexer) is simple
  - compact and cheap
  - but allows for partial reconstruction of state

Building stronger stream ciphers

(1) Introduce non-linearity in state updating function
- irregular clocking: let # LFSR cycles depend on state bit values
- make recursion formula non-linear, e.g., NLFSR

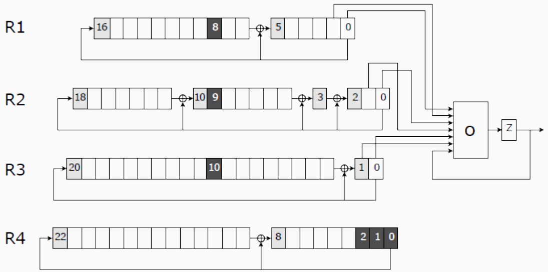
(2) After writing D and K in state, do *blank cycles* (no output)
- non-linearity from D and K to s_t is weak for small t
- but increases fast with growing t
- note: requires state updating function to be non-linear

(3) Make output function stronger
- *research* has led to many published criteria
- choose an output function guided by these

Alternative approach: build stream cipher from a strong cryptographic primitive, e.g., a block cipher or a cryptographic permutation

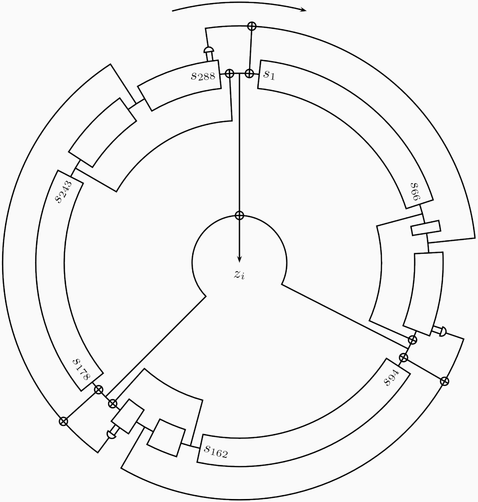
Some real-world stream ciphers

Irregularly clocked LFSR: DECT Stream Cipher



- In use in hundreds of millions of wireless phones today
- 4 maximum-length LFSRs with coprime lengths
- Top 3 clocked 2 or 3 times in between time steps t
- Bottom LFSR determines clocking of top 3 ones
- Output function O with 1 bit of *memory*
- Practically broken with statistical key recovery attack

NLFSR: Trivium [De Canniere and Preneel, 2006]



- claims 80 bits of security
- 80-bit *K* and 80-bit *D*
- 288-bit state
- linear output function
- regularly clocked
- non-linearity in update: only 3 AND gates
- output period not known in advance *but likely OK*
- init. takes 1152 cycles
- as yet unbroken

Something completely different: RC4 [Ron Rivest, 1987]

- 5 to 256-byte K, no dedicated D
- State: 256-byte array, 2 pointers
- Software-oriented
- Used in TLS and WEP
- Biases in keystream
- Broken in practice

[wikipedia]

Key Schedule Algorithm (KSA), initialization:

```
for i from 0 to 255
    S[i] := i
endfor
j := 0
for i from 0 to 255
    j := (j + S[i] + key[i mod keylength]) mod 256
    swap values of S[i] and S[j]
endfor
```

Pseudo-random generation algorithm (PRGA), update/output:

```
i := 0
j := 0
while GeneratingOutput:
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    swap values of S[i] and S[j]
    K := S[(S[i] + S[j]) mod 256]
    output K
endwhile
```

Native stream ciphers: summary

- Filtered and combiner LFSR: mostly of historical significance
  - from days before *resync attacks*
- Irregularly clocked still in use today:
  - DECT, Mifare Classic, GSM A5/1, ...
  - each and every one of them is broken
- So-called *state-of-the-art*: eSTREAM portfolio
  `http://www.ecrypt.eu.org/stream/`
  - NLFSR: Trivium, Grain, Snow and Sosemanuk
  - RC4-inspired design: HC-128

  by academics, for academics
- Reality check:
  - stream encryption in practice today: modes of block ciphers
  - future of stream encryption: modes of permutations

  (both treated later in this course)

Modelling attacks on stream ciphers

Stream cipher definition, recap

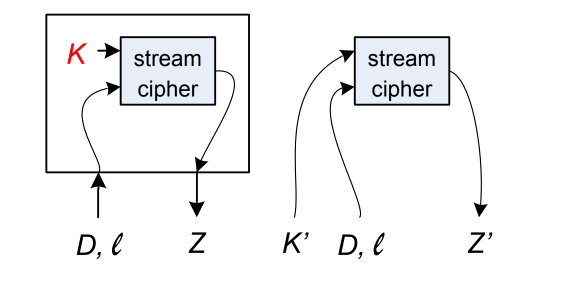
$$K \longrightarrow \boxed{\text{Stream cipher}} \longrightarrow z_1 z_2 z_3 \cdots$$
$$D \longrightarrow$$

- Generates keystream bits z_t from
  - K: secret key, typically 128 or 256 bits
  - D: diversifier, for generating multiple keystreams per key
- z_t can be a bit or a sequence of bits, e.g., a byte or a 32-bit word

We can formally write (asking for ℓ output bits):

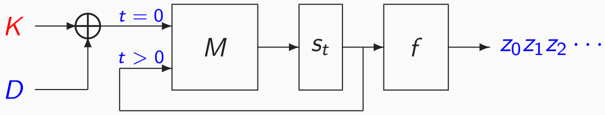
$$Z = \mathrm{SC}_K(D, \ell)$$

Modelling attacks



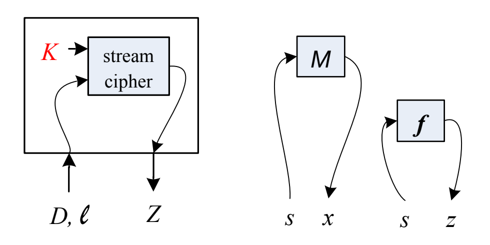
- Adversary has query access to:
  - SC_K: stream cipher instance with unknown key K
  - $\mathsf{SC}_{K'}$: stream cipher instance with chosen key K'
- Can make queries Q
  - Q_d: queries to SC_K with cost (e.g., total length) M
  - Q_c: queries to $\mathsf{SC}_{K'}$ with cost N
- Express probability of success as function of M and N
- Example: generic exhaustive key search: $\Pr(\text{success}) = N2^{-|K|}$ with N number of key-trial queries to $\mathsf{SC}_{K'}$

Iterative stream ciphers: internal structure



- Operates on an evolving state s_t
- In our multiplexer LFSR example:
  - State update function $s_t = Ms_{t-1}$: LFSR update
  - Output function $z_t = f(s_t)$: multiplexer
- More in general
  - State update function $s_t = M(s_{t-1})$
  - Output function: $z_t = f(s_t)$

Modelling attack (cont'd)



- Limitation of previous model: can only describe *generic* attacks
  - generic means: making abstraction of the inner working
- We give adversary query access to inner functions (here M and f)
  - models the fact that these are *public* (Kerckhoff's principle)
- Breakdown can be even more fine-grained
  - queries to inner functions finally become *computation*
  - . . . with some measure of computational effort
- When do we consider a stream cipher *secure*?
  - if no attacks with success probability above the one in claim!
  - but what would be reasonable to claim in the first place?

The Random Oracle

The ideal cipher: Random Oracle [Bellare-Rogaway 1993]

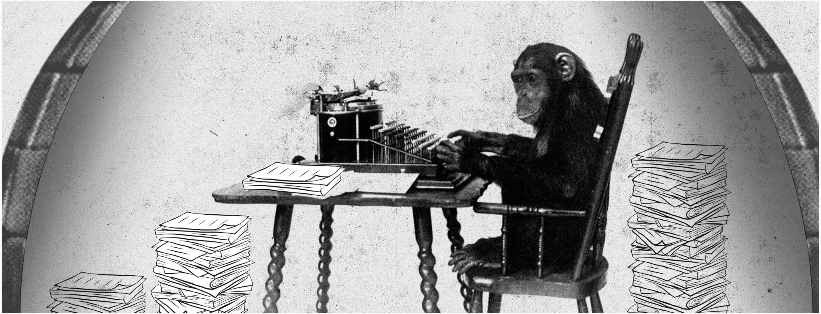
- What would the ideal cryptographic function look like?
- It is called a Random Oracle (RO)
- Random Oracle can be built, but is not practical

Random Oracle Inc.: letter answering service!

Random Oracle Inc.

1) (m, ℓ) arrives at Random Oracle Inc., with
   - m: the letter
   - ℓ: the length of the requested answer

2) Frank checks archive for presence of a file (m, Z)



3) Employee Cheetah is put at work
   a) if no (m, Z) in archive, Cheetah types random Z with $|Z| = \ell$
   b) else if $|Z| < \ell$, Cheetah extends Z to length ℓ with random string
   c) else Cheetah takes a break and eats a banana

4) Frank copies Z

5) Frank puts file with (m, Z) (back) in archive

5) Frank sends response Z truncated to length ℓ to sender

Random Oracle returns unrelated responses for different inputs m

Random Oracle (bit more formal)

- Database of input-output tuples
- Initially empty

m	Z
1100	101011101010101
1111010101101101	11010111011111101101
001000011100	10101101011101010101011
...	...

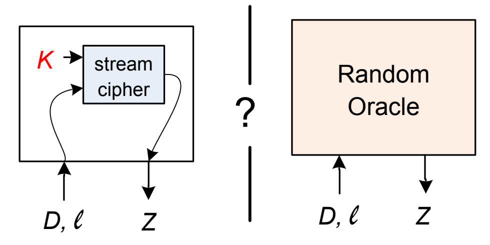
- New query (m, ℓ):
  - If m is not in the database:
    - generate ℓ random bits Z,
    - add (m, Z) to the list,
    - return Z
  - If m is in the database, look at corresponding Z:
    - If $|Z| \geq \ell$: return first ℓ bits of Z
    - If $|Z| < \ell$: generate $\ell - |Z|$ random bits Z',
      append Z' to Z,
      replace (m, Z) in the list by $(m, Z\|Z')$,
      return $Z\|Z'$.

Stream encryption using a random oracle

- Say:
  - Alice wants to send messages P_i confidentially to Bob
  - both Alice and Bob can query $\mathcal{RO}$, but nobody else can
- For message P_i:
  - Alice queries $\mathcal{RO}$ with $(i, |P_i|)$ and $\mathcal{RO}$ returns Z_i
  - Alice enciphers P_i to $C_i \leftarrow P_i + Z_i$
  - Alice sends (i, C_i) to Bob
  - Bob queries $\mathcal{RO}$ with $(i, |C_i|)$ and $\mathcal{RO}$ returns Z_i
  - Bob deciphers C_i to $P_i \leftarrow C_i + Z_i$
- The $\mathcal{RO}$ returns a one-time pad, so provides perfect secrecy
- If we have stream cipher that, if keyed, is indistinguishable from $\mathcal{RO}$
  - Alice and Bob use that to get Z_i instead of $\mathcal{RO}$ and it's OK!
  - that suggests a useful security goal for a stream cipher

Indistinguishability security notion

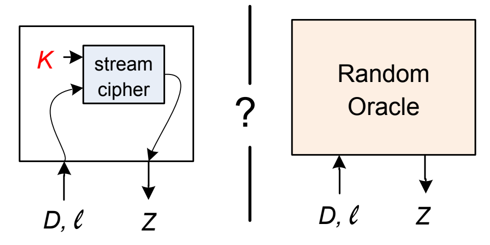
Security notion: hardness of distinguishing from $\mathcal{RO}$



Distinguishing game (black box version):

- Adversary $\mathcal{A}$ has query access to a system that is either:
  - SC_K: stream cipher with unknown key K
  - $\mathcal{RO}$: ideal stream cipher in the form of a random oracle
- She does not know which one and has to guess that
- Adversary $\mathcal{A}$ is actually an *attack algorithm* that returns either:
  - 1 if it estimates the system is SC_K
  - 0 if it estimates the system is $\mathcal{RO}$

Hardness of distinguishing from $\mathcal{RO}$: advantage



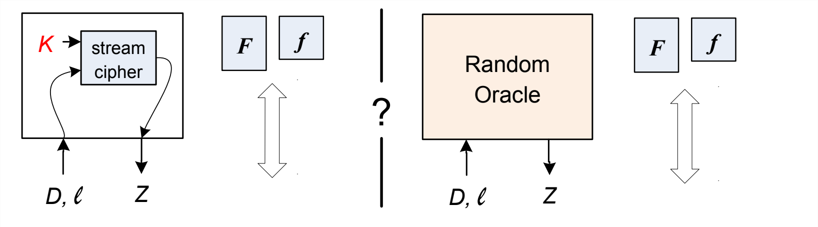
- Let:
  - $\mathsf{Pr}(\mathcal{A} = 1 \mid \mathsf{SC}_K)$: probability that $\mathcal{A}$ returns 1 in case of SC_K
  - $\mathsf{Pr}(\mathcal{A} = 1 \mid \mathcal{RO})$: probability that $\mathcal{A}$ returns 1 in case of $\mathcal{RO}$

Advantage of an adversary $\mathcal{A}$

$$\mathrm{Adv}_{\mathcal{A}} = |\mathsf{Pr}(\mathcal{A} = 1 \mid \mathsf{SC}_K) - \mathsf{Pr}(\mathcal{A} = 1 \mid \mathcal{RO})|$$

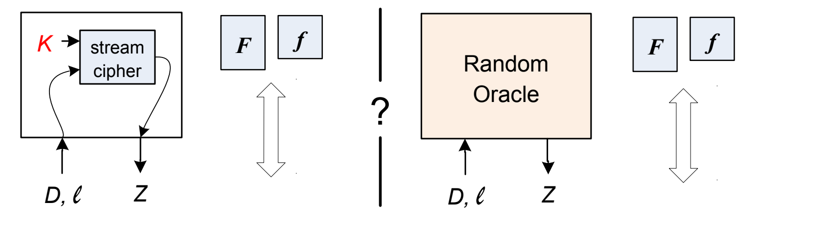
Note: $\mathrm{Adv}_{\mathcal{A}}$ is in interval $[0 \ldots 1]$

Hardness of distinguishing from $\mathcal{RO}$: resources



- Black box fails to model that F and f are public
- We give additional query access to F and f
- We model query complexity in two parts again:
  - M: called online or data complexity
  - N: called offline or computational complexity
- We express $\mathrm{Adv}_{\mathcal{A}}$ as $\epsilon(M, N)$

Hardness of distinguishing from $\mathcal{RO}$: what a claim looks like



$\epsilon(M, N)$ indistinguishability claim for a stream cipher SC

There exists no attack algorithm $\mathcal{A}$ that distinguishes SC_K, with K a uniformly chosen unknown key, from a random oracle with $\text{Adv}_{\mathcal{A}} > \epsilon(M, N)$

Note: this is a very powerful type of claim

Implications of $\mathcal{RO}$ indistinguishability

A $\epsilon(M, N)$ indistinguishability claim implies:

- There are no key recovery attacks with success prob. above $\epsilon(M, N)$
- Probability that keystream Z is periodic is below $\epsilon(M, N)$
- Success in exploiting biases in Z limited to $\epsilon(M, N)$
- ...

Implications of a $\epsilon(M, N)$ indistinguishability claim

It claims for any imaginable attack:

$$\Pr(\text{success of attack on } \mathrm{SC}_K) \leq \epsilon(M, N) + \Pr(\text{success of attack on } \mathcal{RO})$$

Proof:

- Recipe for distinguishing adversary $\mathcal{A}$ based on the attack:
  1. Spend resources M and N on the attack
  2. If it works, return 1, else return 0
- $\Pr(\mathcal{A} = 1 \mid \mathcal{RO}) = \Pr(\text{success of attack on } \mathcal{RO})$
- $\Pr(\mathcal{A} = 1 \mid \mathrm{SC}_K) = \Pr(\text{success of attack on } \mathrm{SC}_K)$
- Due to the claim their difference is at most $\epsilon(M, N)$

Conclusion: what is a secure stream cipher?

- A stream cipher that, when keyed with a fixed and unknown key K, is hard to distinguish from a random oracle
- How hard it actually is, is expressed by a bound on the advantage
- We cannot prove such bounds for concrete stream ciphers
- But we can make claims and assumptions
- For stream ciphers built on an underlying primitive we can prove conditional bounds ...