رمزنگاری، امنیت اطلاعات و حریم خصوصی

ارائه: دکتر سیدعلی لاجوردی

بخش ششم

So far…

- We have seen how to construct schemes based on various lower-level primitives
  - Stream ciphers/PRGs
  - Block ciphers/PRFs
  - Hash functions (compression functions)

- How do we construct these primitives?

Two approaches

- Construct from even lower-level assumptions
  - Can prove secure (given lower-level assumption)
  - Inefficient

- Build directly
  - Much more efficient!
  - Need to assume security, but…
    - We have formal definitions to aim for
    - We can concentrate our analysis on these primitives
    - We can develop/analyze various design principles

Stream ciphers/PRGs

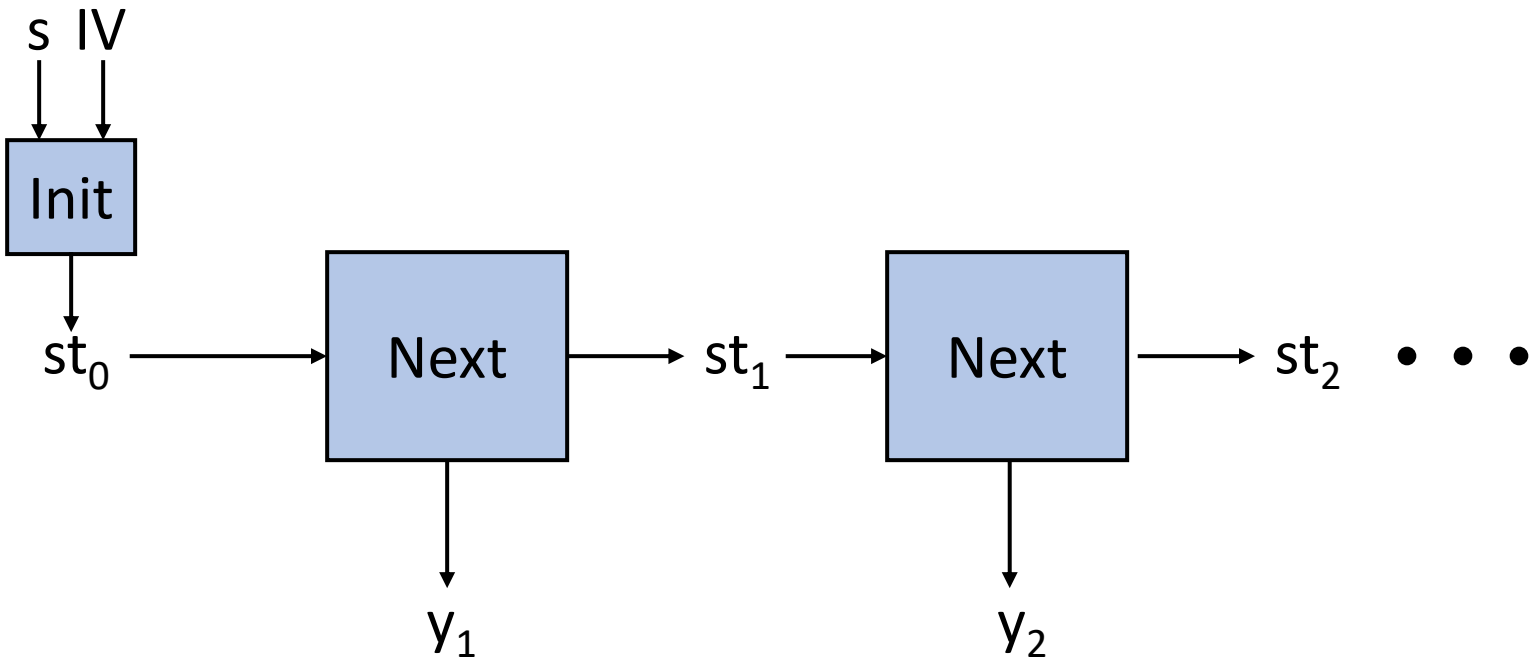
Terminology

- Init algorithm
  - Takes as input a key [+ initialization vector (IV)]
  - Outputs initial state

- Next algorithm
  - Takes as input the current state
  - Outputs next bit/byte/chunk and updated state
  - Allows generation of as many bits as needed

Stream ciphers

- Can use (Init, Next) to generate any desired number of output bits from an initial seed



Security requirements

- If there is no IV, then (for a uniform key) the output of Next should be indistinguishable from a uniform, independent stream of bits

- If there is an IV, then (for a uniform key) the outputs of Next on multiple, uniform IVs should be indistinguishable from multiple uniform, independent streams of bits
  - Even if the attacker is given the IVs

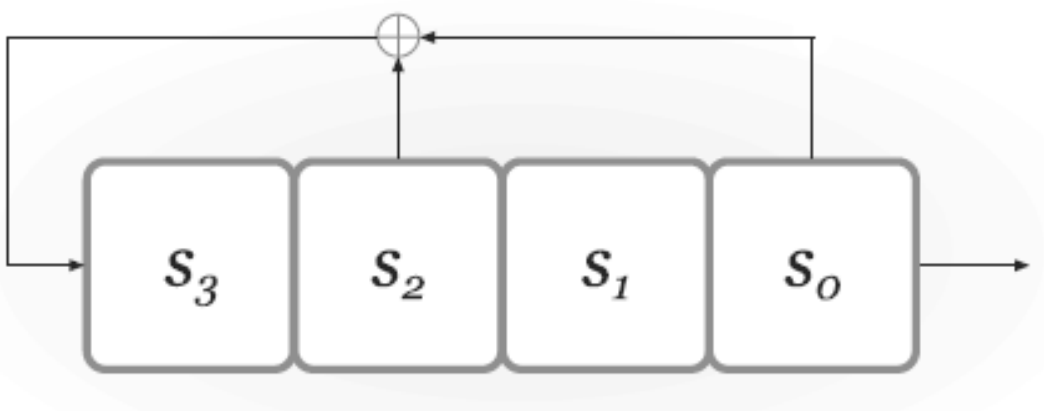
Security requirements

- In practice, want near-optimal concrete security
  - Not just asymptotic security

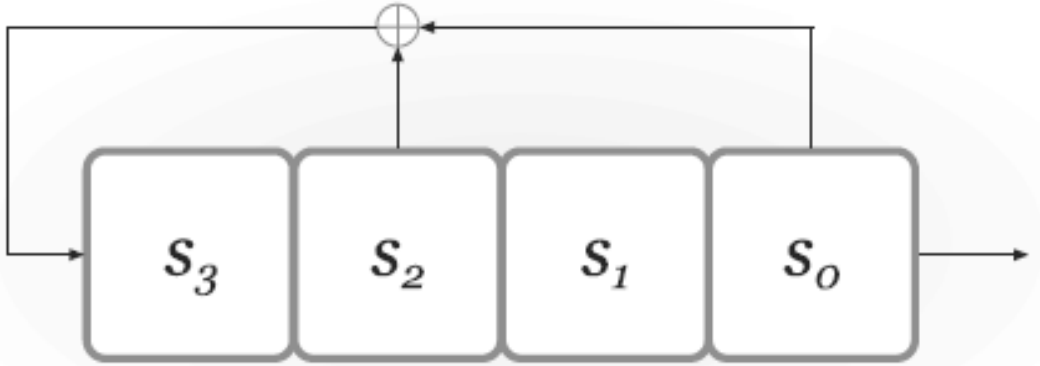
- Stream cipher with n-bit key should be secure against attackers running in time $\approx 2n$

LFSRs

- Degree n $\Rightarrow$ n registers
- State: bits sn-1, …, s0 (contents of the registers)
- Feedback coefficients cn-1, …, c0 (do not change; part of the design, not the state)
- Registers updated, and output generated, in each “clock tick”



Example



- Assume initial content of registers is 0100
- First 4 state transitions:

  $0100 \rightarrow 1010 \rightarrow 0101 \rightarrow 0010 \rightarrow \ldots$
- First 3 output bits:

  0 0 1 …

LFSRs as stream ciphers

- Key (+ IV) used to initialize state of the LFSR (possibly including feedback coefficients)

- One bit of output per clock tick
  - State is updated each clock tick

LFSRs

- State (and output) “cycles” if state ever repeated
  - Short cycles are bad for security
  - How long can a cycle be?

- A maximal-length LFSR cycles through all 2n - 1 nonzero states
  - It is known how to set feedback coefficients so as to achieve maximal length

- Maximal-length LFSRs have good statistical properties…
- …but they are not cryptographically secure!

Security?

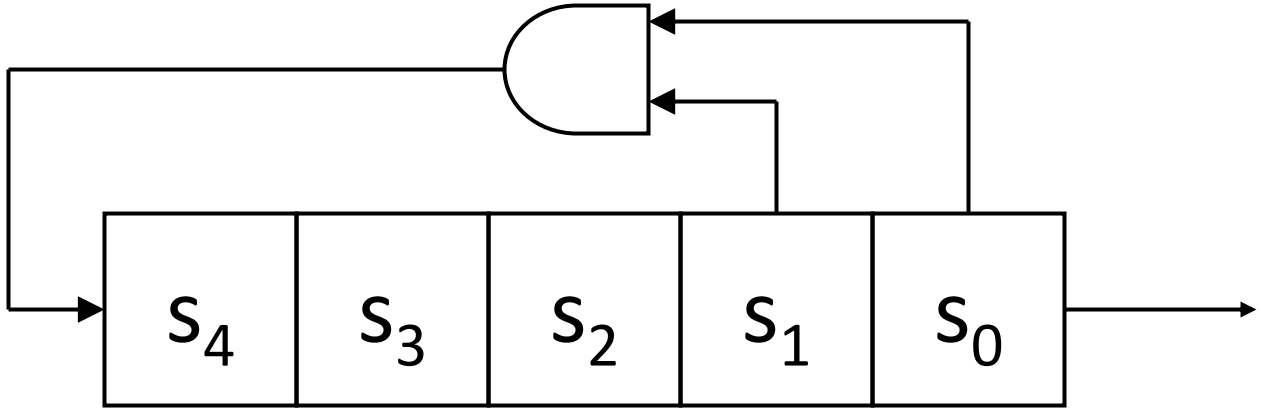
- If feedback coefficients are fixed (and hence known to the attacker), then the key just determines the initial register contents
- First n bits of the output reveal the entire key!
- Even if feedback coefficients are unknown (and determined by the key), can use linear algebra to learn everything from initial 2n output bits
- Moral: linearity is bad for pseudorandomness (because linear algebra is so powerful)

Nonlinear FSRs

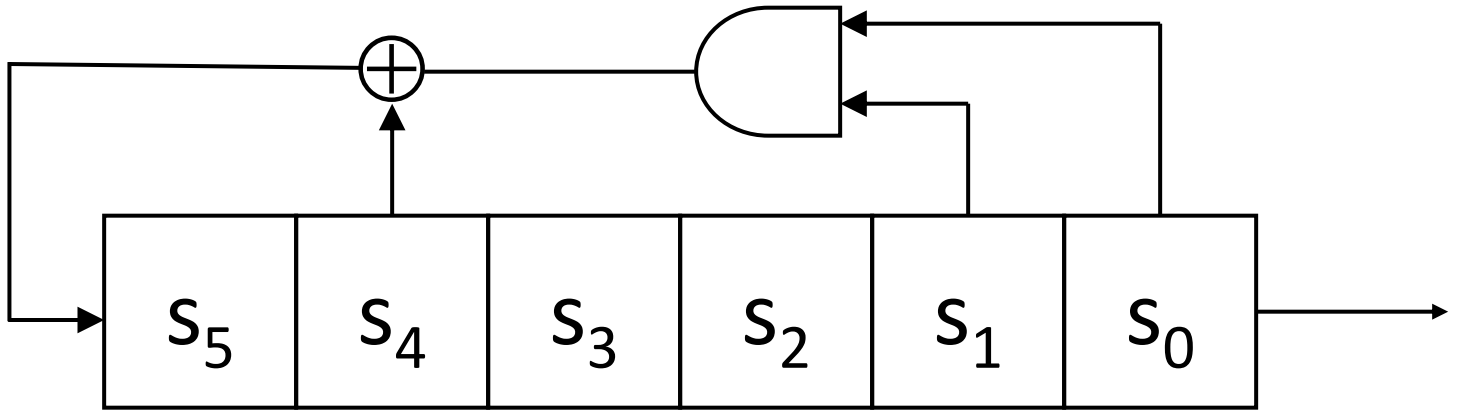
- Add nonlinearity to prevent attacks
  - Nonlinear feedback
  - Nonlinear output (nonlinear filter)
  - Multiple LFSRs (combination generator)
  - …or some combination of the above
- Still want to preserve statistical properties of the output, and long cycle length
- From now on, assume design (including feedback coefficients) is fixed
  - Key only determines the initial register contents

Nonlinear feedback

Nonlinear feedback

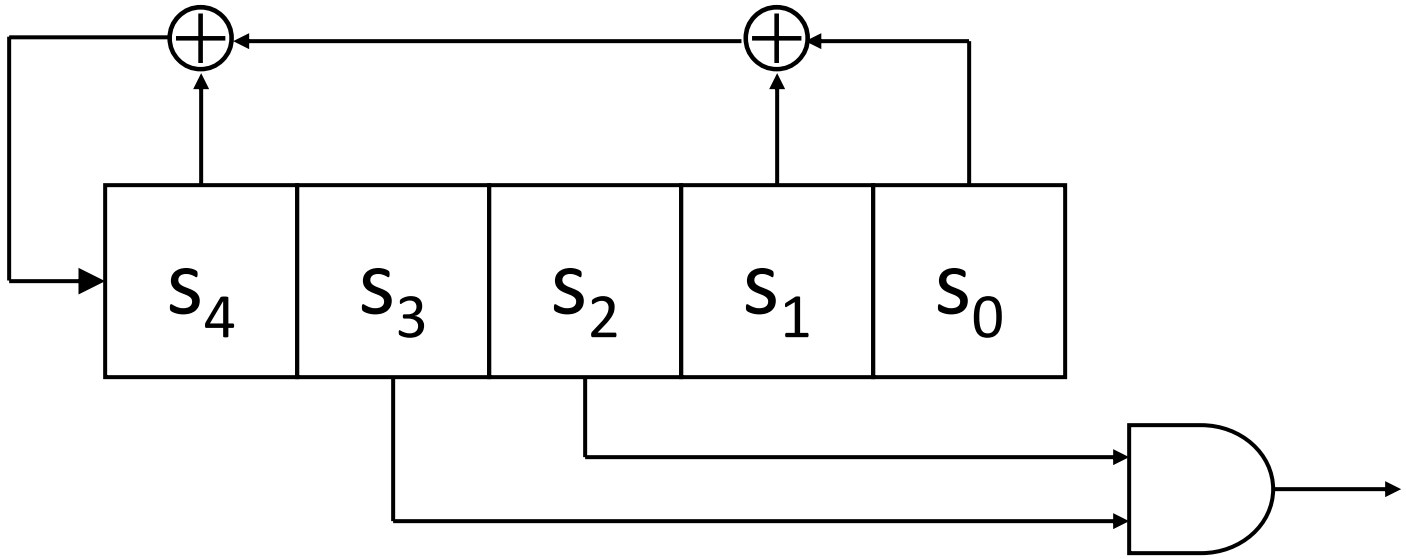


- Need to avoid bias!



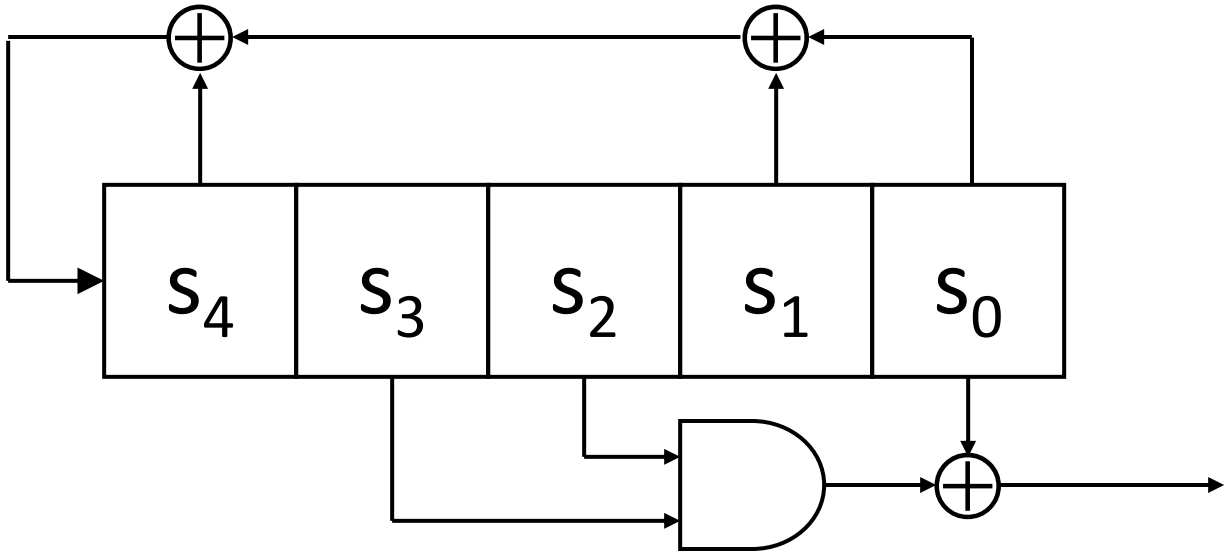
Nonlinear filter

- Update of state is still linear…
- …but output is a nonlinear function of the entire state

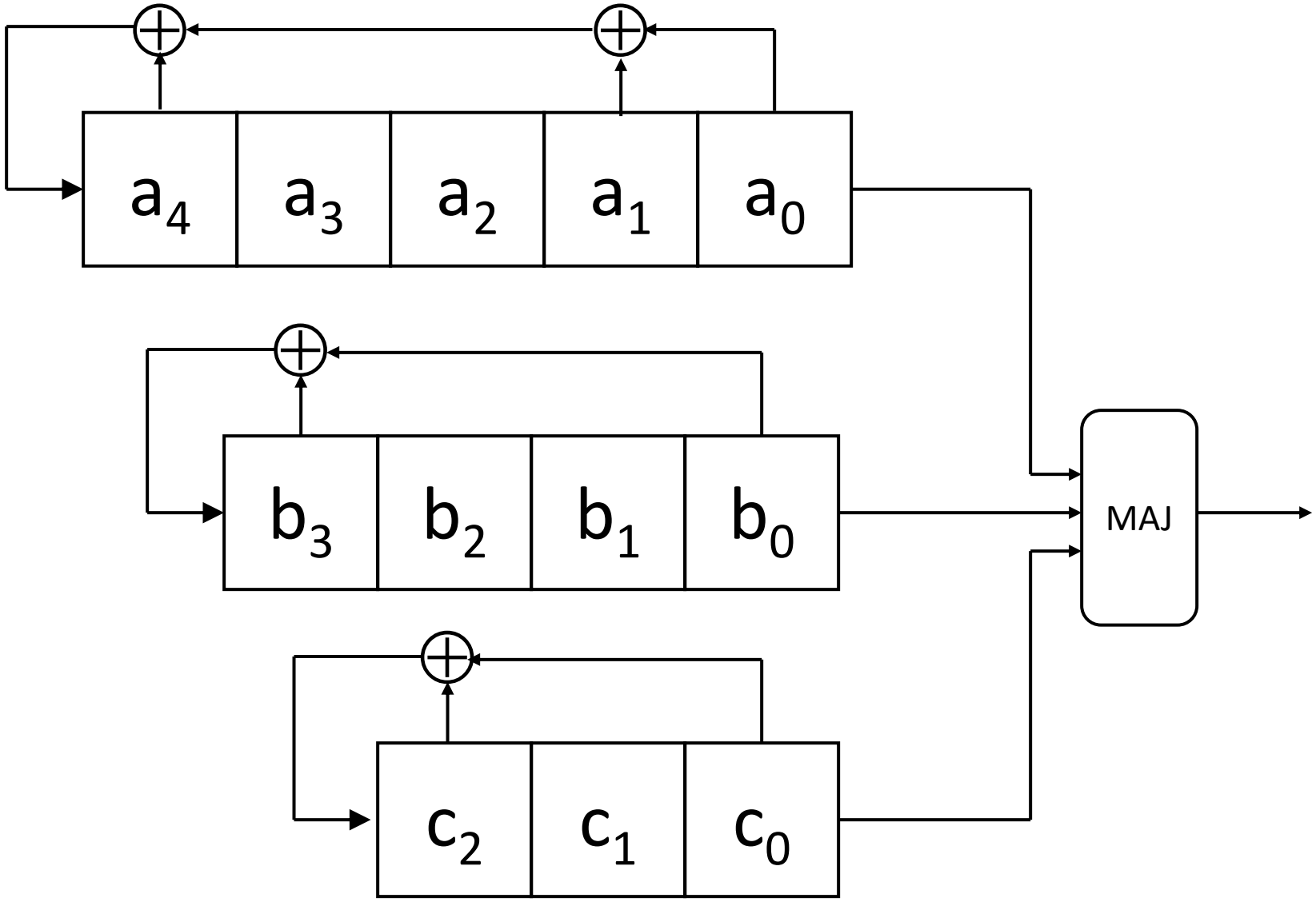


Nonlinear filter

- Need to avoid bias!



Combination generator



Correlation attacks

- Consider previous example, and let A, B, and C be the output sequence generated by each LFSR
  - So the overall output is MAJ(A, B, C)

- Let ρ, σ, τ denote the degree of each LFSR
  - Key has length $\rho + \sigma + \tau$
  - Want security for attacks running in time $2\rho + \sigma + \tau$

Correlation attacks

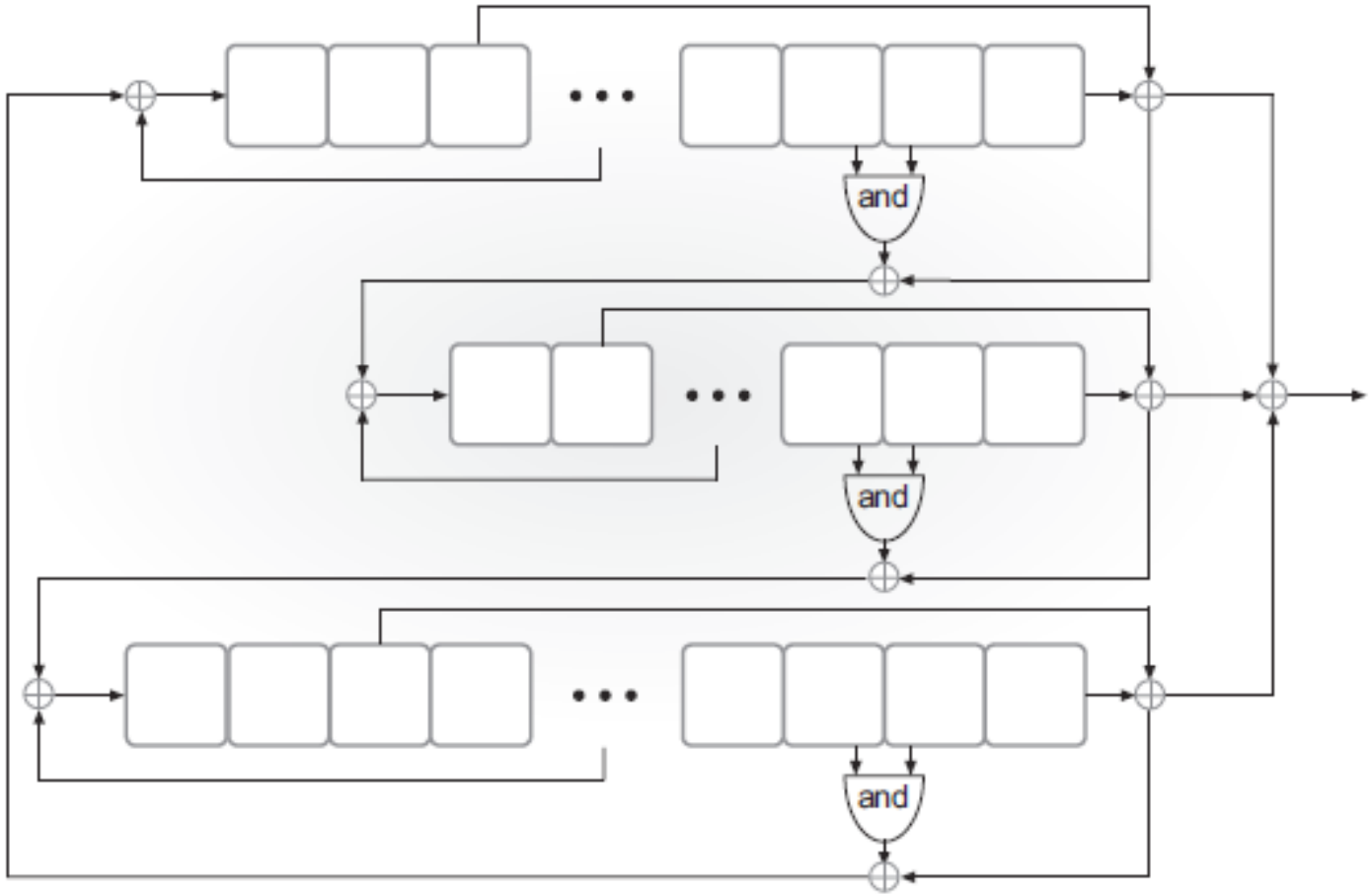
- Key observation: A, B, and C are each highly correlated with the output
  - Assuming B, C are unbiased, Pr[Ai = outputi] = ¾ for all i (and similarly for B, C)
  - Alternately, for large enough sequences, ¾ of the bits in R should be equal to the corresponding output bits

- Can do a brute-force search over the state of each LFSR individually!
  - Key-recovery attack runs in time $2\rho + 2\sigma + 2\tau < 2\rho + \sigma + \tau$

Trivium

- Designed by De Cannière and Preneel in 2006 as part of eSTREAM project
- Intended to be simple and efficient (especially in hardware)
- No attacks better than brute-force search are known!

Trivium



Trivium

- Three coupled FSRs of degree 93, 84, and 111
  - 288-bit state
- Initialization:
  - 80-bit key in left-most registers of first FSR
  - 80-bit IV in left-most registers of second FSR
  - Remaining registers set to 0, except for three right-most registers of third FSR
  - Run for 4 x 288 clock ticks (output discarded)

RC4

- Designed in 1987
- Designed to have good performance in software, rather than hardware

- No longer considered secure, but still interesting to study
  - Simple description; not LFSR-based
  - Still encountered in practice
  - Interesting attacks

RC4

- State consists of a 256-byte array S, which is always a permutation of {0,1}8, along with integers $0 \leq i, j \leq 255$
  - Note S can be viewed as a permutation of {0,1}8 that is constantly changing

RC4

- Not designed to take an IV, but often used with an IV anyway
  - E.g., prepend IV to the key

ALGORITHM 6.1
Init algorithm for RC4

Input: 16-byte key k
Output: Initial state (S, i, j)
(Note: All addition is done modulo 256)

for $i = 0$ to 255:
    $S[i] := i$
    $k[i] := k[i \bmod 16]$
$j := 0$
for $i = 0$ to 255:
    $j := j + S[i] + k[i]$
    Swap $S[i]$ and $S[j]$
$i := 0, \quad j := 0$
return (S, i, j)

ALGORITHM 6.2
GetBits algorithm for RC4

Input: Current state (S, i, j)
Output: Output byte y; updated state (S, i, j)
(Note: All addition is done modulo 256)

$i := i + 1$
$j := j + S[i]$
Swap $S[i]$ and $S[j]$
$t := S[i] + S[j]$
$y := S[t]$
return $(S, i, j), y$

Attack: bias in 2nd output byte

- Let St denote permutation S after t steps
  - Treat S0 as uniform for simplicity
- Say X = S0[1] $\neq$ 2 and S0[2] = 0
  - Occurs with probability $\approx$1/256
  - Then:
    - After 1 step, S1[X]=X, i=1, j=X
    - After 2 steps, j=X; output S2[X]=0
- Otherwise, S2[X] is a uniform byte
- Pr[2nd byte is 0] $\approx$2/256

RC4 bias

- Statistical bias in other output bytes was determined experimentally

- Enough to break pseudo-OTP encryption based on RC4!
  - See http://www.isg.rhul.ac.uk/tls

Block ciphers

Recall…

- Want keyed permutation

  $F: \{0,1\}^n \times \{0,1\}^l \rightarrow \{0,1\}^l$

  - n = key length, l = block length

- Want Fk (for uniform, unknown key k) to be indistinguishable from a uniform permutation over $\{0,1\}^l$, for attacks running in time $\approx 2^n$

Attack models

- A block cipher is not an encryption scheme!!
- Nevertheless, some of the terminology used is the same (for historical reasons)
  - “known-plaintext attack”: attacker given {(x, Fk(x)} for arbitrary x (outside control of the attacker)
  - “chosen-plaintext attack”: attacker can query Fk(.)
  - “chosen-ciphertext attack”: attacker can query both Fk(.) and Fk-1(.)

Concrete security

- As in the case of stream ciphers, we are interested in concrete security for a given key length n
  - Best attack should take time $\approx 2^n$
  - If there is an attack taking time $2^{n/2}$ then the cipher is considered insecure

- Look at both distinguishing attacks and key-recovery attacks

Designing block ciphers

- Want Fk (for uniform, unknown key k) to be indistinguishable from a uniform permutation over {0,1}l
- If x and x’ differ in one bit, what should the relation between Fk(x) and Fk(x’) be?
  - How many bits should change (on average)?
  - Which bits should change?
- How to achieve this?

Confusion/diffusion

- Two types of steps
  - “Confusion”: Small change in input to the step yields small, “random” change in output of the step
  - “Diffusion”: Small change in input to the step should be propagated to affect entire output of the step

Design paradigms

- Two design paradigms
  - Substitution-permutation networks (SPNs)
  - Feistel networks

SPNs

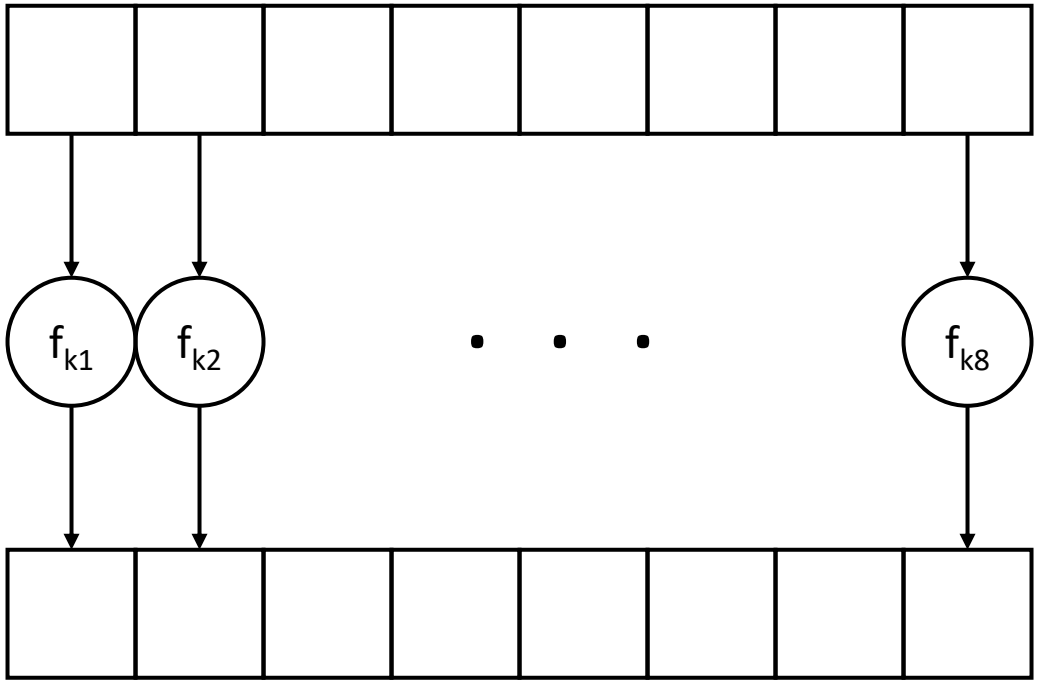
SPNs

- Build “random-looking” permutation on long input from random permutations on short input
  - What is the key length for a random permutation on {0,1}l ?

- E.g. (assuming 8-byte block length),
  Fk(x) = fk1(x1) fk2(x2) … fk8(x8),
  where each f is a random permutation on {0,1}8
  - How long is the key for F?

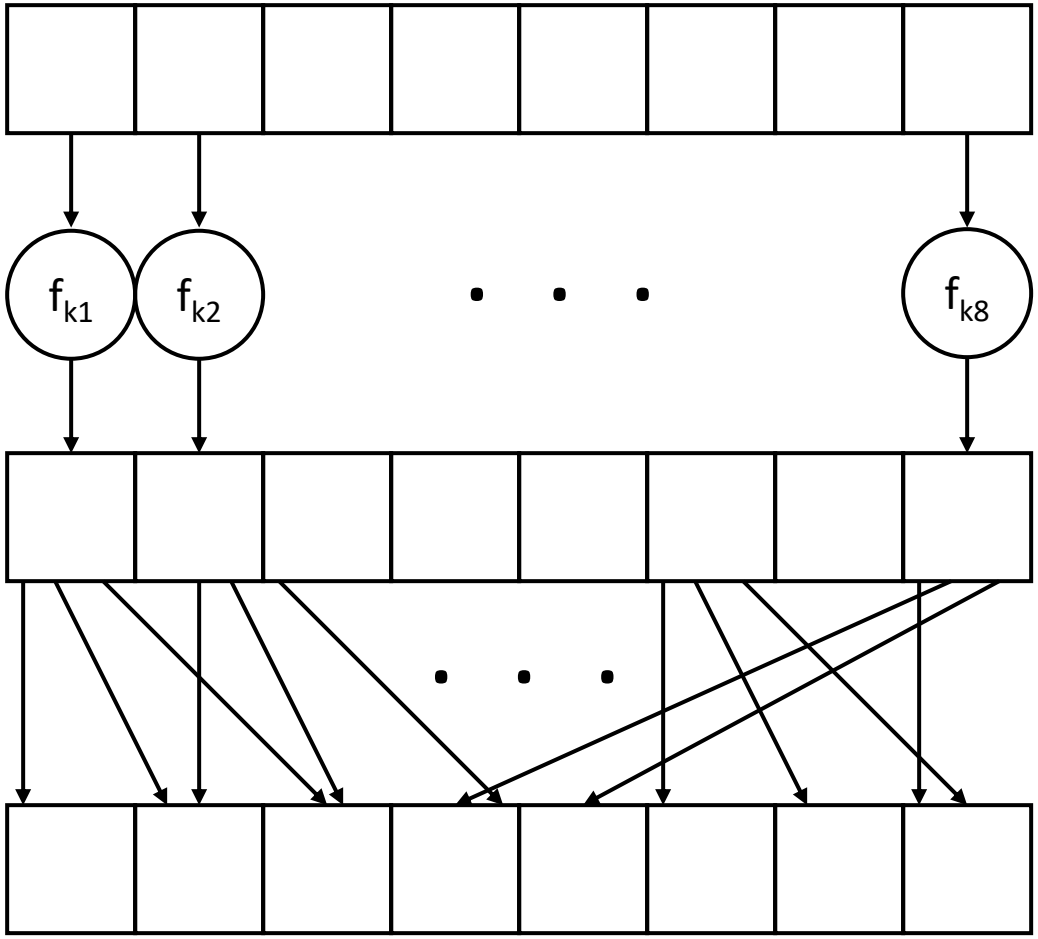
SPN

- This has confusion but no diffusion
  - Add a mixing permutation…



Is this a pseudorandom permutation?

SPN



SPN

- Mixing permutation is public/known to the attacker
  - Chosen to ensure good diffusion
  - (This will be more clear later)

- Note that the entire structure is invertible (given the key) since the f's are permutations and the mixing permutation is invertible

SPN

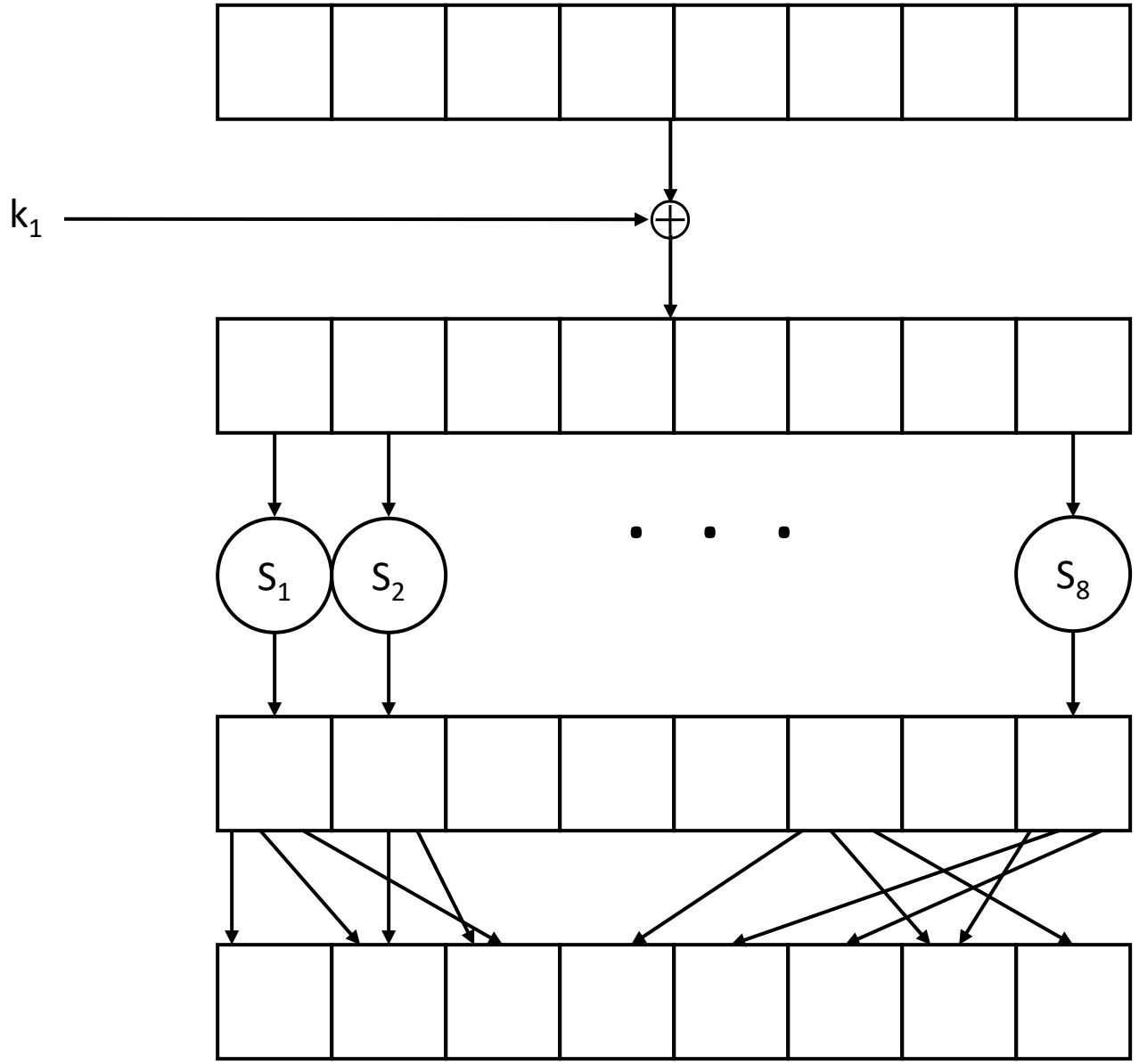
- Does this give a pseudorandom permutation?

- What if we repeat for another round (with independent, random functions)?
  - What is the minimal # of rounds we need?
  - Avalanche effect
  - Judicious choice of mixing permutation

SPNs

- Using random f's is not practical
  - Key would be too large

- Instead, use f's of a particular form
  - fki(x) = Si(ki $\oplus$ x), where Si is a fixed (public) permutation
  - The {Si} are called "S-boxes" (substitution boxes)
  - XORing the key is called "key mixing"
  - Note that this is still invertible (given the key)


k1
S1
S2
S8

Avalanche effect

- Design S-boxes and mixing permutation to ensure avalanche effect
  - Small differences should eventually propagate to entire output
- S-boxes: any 1-bit change in input causes ≥2-bit change in output (confusion)
  - Not so easy to ensure!
- Mixing permutation
  - Each bit output from a given S-box should feed into a different S-box in the next round (diffusion)

SPN

- One round of an SPN involves
  - Key mixing
    - Round keys could be independent
    - In practice, derived from a master key via a key schedule
  - Substitution (S-boxes)
  - Permutation (mixing permutation)
- r-round SPN has r rounds as above, plus a final key-mixing step
  - Why?
- Invertible regardless of how many rounds…

Key-recovery attacks

- Key-recovery attacks are even more damaging than distinguishing attacks
  - As before, a cipher is secure only if the best key-recovery attack takes time $\approx 2^n$
  - A fast key-recovery attack represents a “complete break” of the cipher

Key-recovery attack, 1-round SPN

- Consider first the case where there is no final key-mixing step
  - Possible to get the key immediately!

- What about a full 1-round SPN (with independent round keys)?
  - Attack 1: for each possible 1st-round key, get corresponding 2nd-round key
    - Continue process of elimination using additional plaintext/ciphertext pairs
    - Complexity $\approx 2^l$ for key of length 2^l
- Better attack: work S-box-by-S-box
  - Assume 8-bit S-box
  - For each 8 bits of 1st-round key, get corresponding 8 bits of 2nd-round key
    - Continue process of elimination
    - Complexity?

Attacking more rounds?

- These attacks become more and more difficult as the number of rounds increases
- At some point, key-recovery attacks become impractical
  - Distinguishing attacks may still be possible, especially if S-boxes are poorly designed
- 3-round SPNs can be proven secure when S-boxes are modeled as random permutations

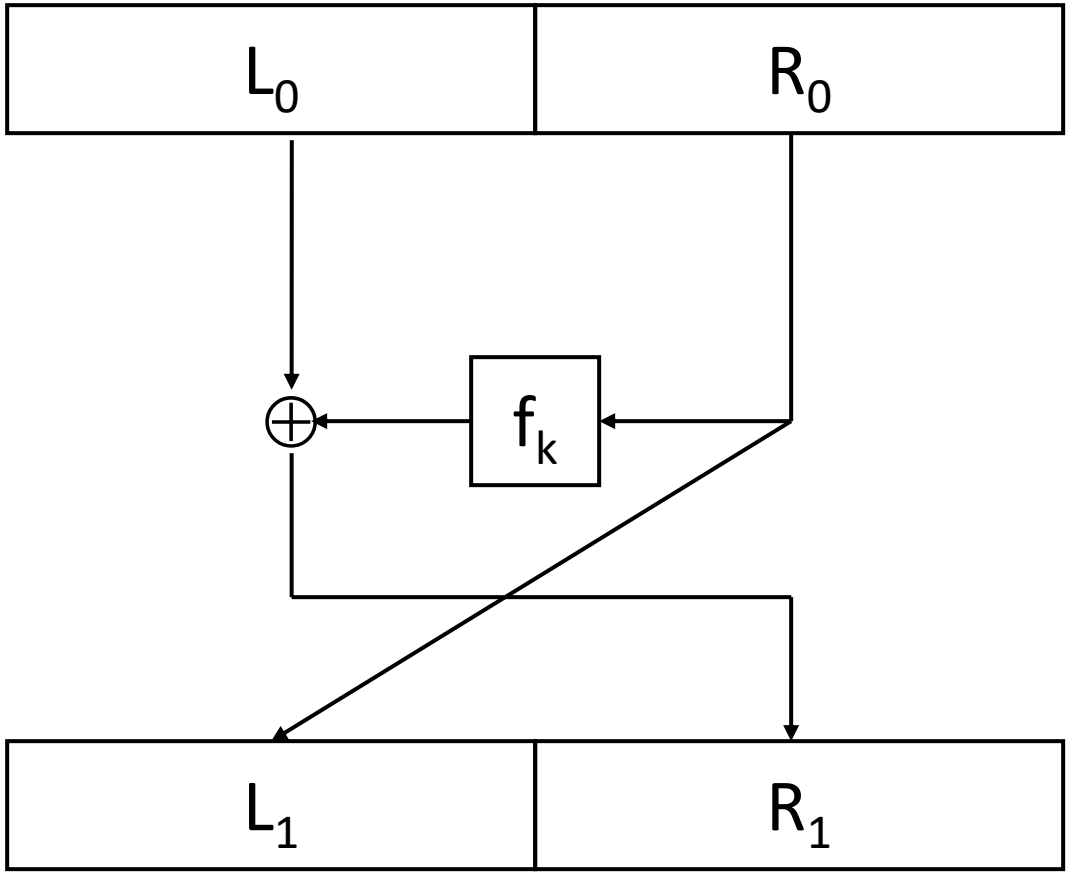
Feistel networks

Feistel networks

- Build (invertible) permutation from non-invertible components

- One round:
  - Keyed round function f: {0,1}n x {0,1}l/2 $\rightarrow${0,1}l/2
  - Fk1(L0, R0) $\rightarrow$ (L1, R1) = (R0, L0 $\oplus$ fk1(R0))

- Always invertible!


L_0
R_0
f_k
L_1
R_1

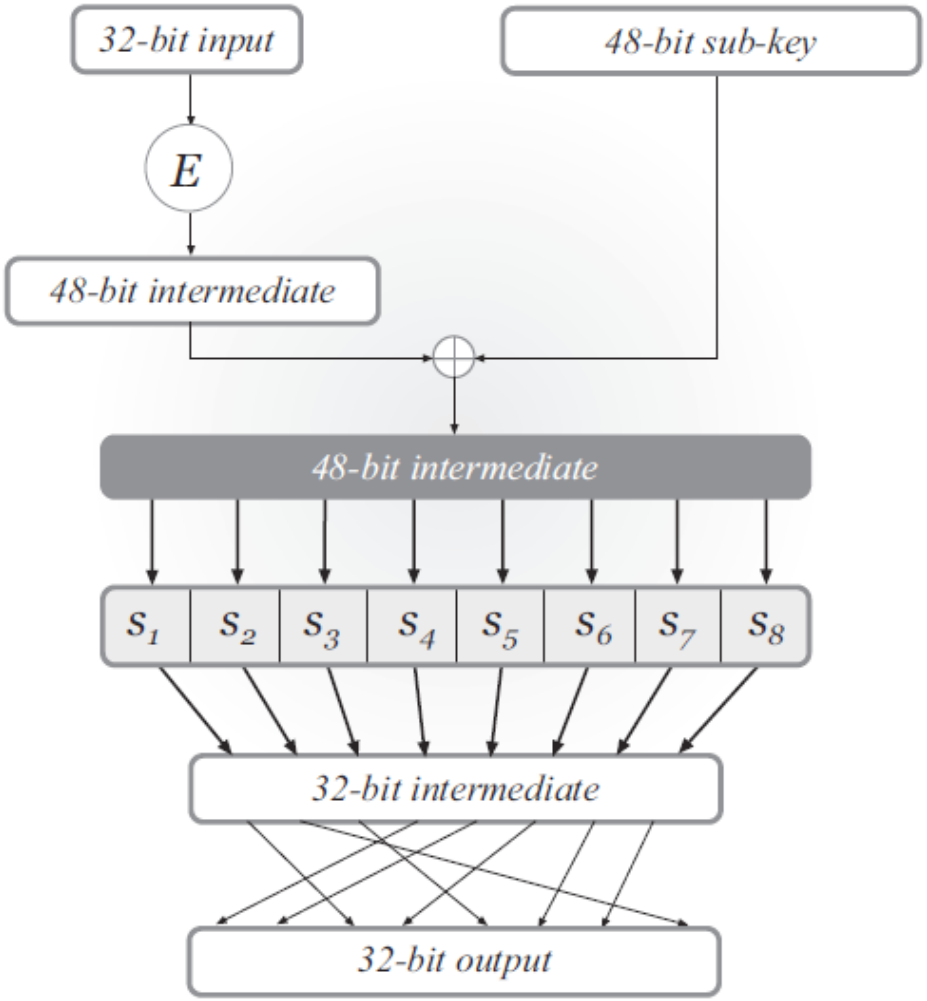
Security?

- Security of 1-round Feistel?
- Security of 2-round Feistel?
- Security of 3/4-round Feistel?
  - (Assume round functions are random and independent)

Data Encryption Standard (DES)

- Standardized in 1977
- 56-bit keys, 64-bit block length
- 16-round Feistel network
  - Same round function (“mangler function”) in all rounds
  - Different sub-keys in each round, each derived from the master key
  - The round function is basically an SPN!

DES mangler function



DES mangler function

- S-boxes
  - Each S-box is 4-to-1
  - Changing 1 bit of input changes at least 2 bits of output
- Mixing permutation
  - The 4 bits of output from any S-box affect the input to 6 S-boxes in the next round

Key schedule

- 56-bit master key, 48-bit subkey in each round
  - Each subkey takes 24 bits from the left half of the master key, and 24 bits from the right half of the master key

Avalanche effect

- Consider 1-bit difference in left half of input
  - After 1 round, 1-bit difference in right half
  - S-boxes cause a 2-bit difference, implying a 3-bit difference overall after 2 rounds
  - Mixing permutation spreads differences into different S-boxes
  - …

Security of DES

- DES is extremely well-designed
  - Except for some attacks that require large amounts of plaintext, no attacks better than brute-force are known

- But … parameters are too small!

56-bit key length

- A concern as soon as DES was released

- Brute-force search over 256 keys is possible
  - 1997: 1000s of computers, 96 days
  - 1998: distributed.net, 41 days
  - 1999: Deep Crack ($250,000), 56 hours
  - Today: 48 FPGAs, ~1 day

64-bit block length

- Birthday collisions relatively likely

- E.g., encrypt 230 (≈ 1 billion) blocks using CTR mode; chances of a collision are

$$\approx 260/264 = 1/16$$

Increasing key length?

- DES has a key that is too short

- How to fix?
  - Design new cipher
  - Tweak DES so that it takes a larger key
  - Build new cipher using DES as a black box

Double encryption

- Let F: {0,1}n x {0,1}l $\rightarrow$ {0,1}l
  - (i.e., n=56, l=64 for DES)
- Define F2 : {0,1}2n x {0,1}l $\rightarrow$ {0,1}l as follows:
  F2k1, k2(x) = Fk1(Fk2(x))
  (still invertible)
- If best attack on F takes time 2n, can we hope that the best attack on F2 takes time 22n?

Meet-in-the-middle attack

- No! There is an attack taking 2n time…
  - (And 2n memory)

- The attack applies any time a block cipher can be “factored” into 2 independent components

Triple encryption

- Define F3 : {0,1}3n x {0,1}l $\rightarrow$ {0,1}l as follows:
  F3k1, k2, k3(x) = Fk1(Fk2(Fk3(x)))

- What is the best attack now?

Two-key triple encryption

- Define F3 : {0,1}2n x {0,1}l $\rightarrow$ {0,1}l as follows:
  F3k1, k2(x) = Fk1(Fk2(Fk1(x)))

- Best attack takes time 22n – optimal given the key length!

- This approach is taken by triple-DES

Advanced encryption standard (AES)

- Public design competition run by NIST
- Began in Jan 1997
  - 15 algorithms submitted
- Workshops in 1998, 1999
  - Narrowed to 5 finalists
- Workshop in early 2000; winner announced in late 2000
  - Factors besides security taken into account

AES

- 128-bit block length
- 128-, 192-, and 256-bit key lengths

- Basically an SPN structure!
  - 1-byte S-box (same for all bytes)
  - Mixing permutation replaced by invertible linear transformation
    - If two inputs differ in b bytes, outputs differ in ≥ 5-b bytes
- No attacks better than brute-force known

SHA-2

- Compression function based on Davies-Meyer
  - With “block cipher” specifically designed for SHA

- Hash function built from compression function using Merkle-Damgard

SHA-3

- Public competition run by NIST
- Began in 2007
- Narrowed to 14 semi-finalists in Dec 2008
- Reduced to 5 finalists in 2010
- Winner chosen in Oct 2012

SHA-3

- Supports 224-, 256-, 384-, and 512-bit output lengths
- Very different design than SHA-1/SHA-2
  - Does not use Davies-Meyer
  - Does not use Merkle-Damgard
  - See book for details

Private-key cryptography

Private-key cryptography

- Private-key cryptography allows two users who share a secret key to establish a “secure channel”

- The need to share a secret key has several drawbacks…

The key-distribution problem

- How do users share a key in the first place?
  - Need to share the key using a secure channel…
- This problem can be solved in some settings
  - E.g., physical proximity, trusted courier, …
  - Note: this does not make private-key cryptography useless!
- Can be difficult, expensive, or impossible to solve in other settings

The key-management problem

- Imagine an organization with N employees, where each pair of employees might need to communicate securely

- Solution using private-key cryptography:
  - Each user shares a key with all other users
  - Each user must store/manage N-1 secret keys!
  - O(N2) keys overall!

Lack of support for “open systems”

- Say two users who have no prior relationship want to communicate securely
  - When would they ever have shared a key?

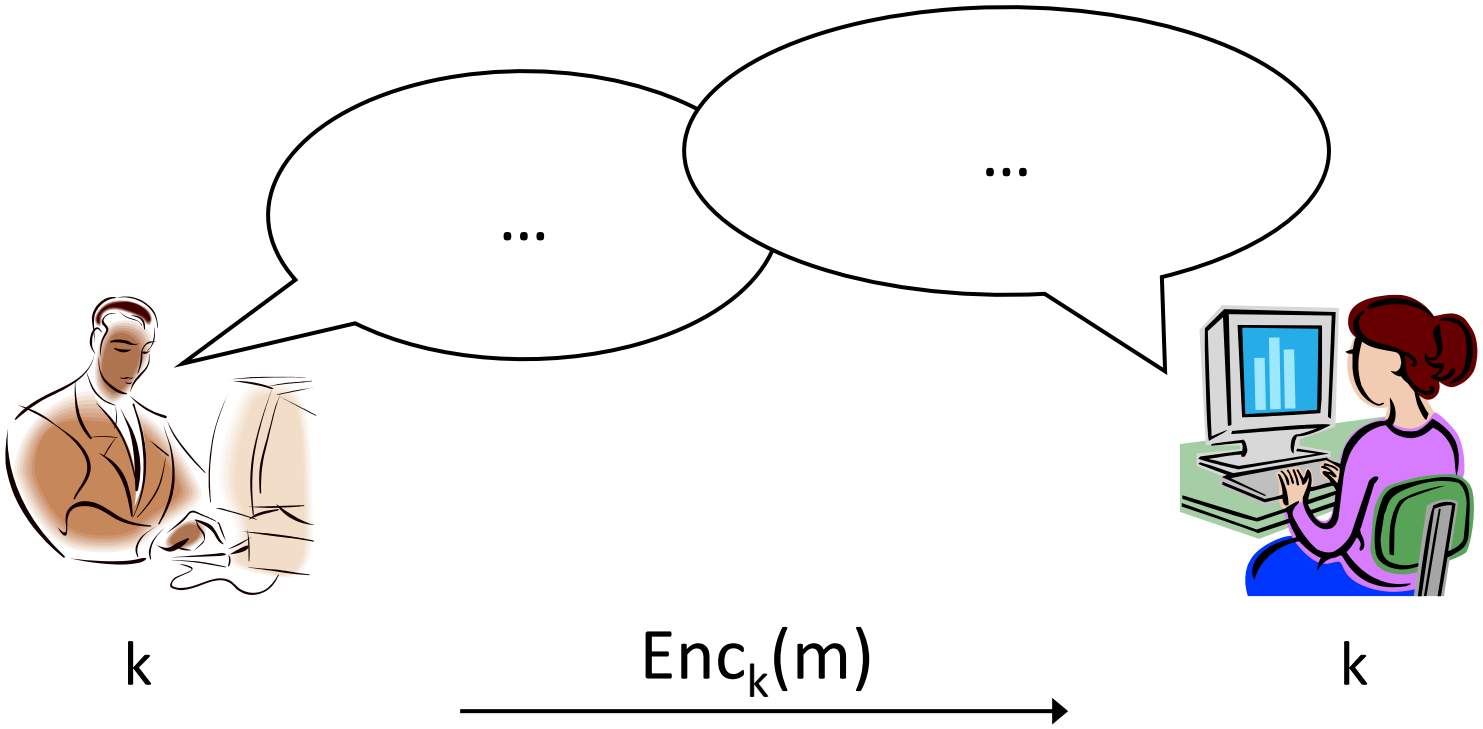
- This happens all the time!
  - Customer sending credit-card data to merchant
  - Contacting a friend-of-a-friend on social media
  - Emailing a colleague

“Classical” cryptography
offers no solution
to these problems!

New directions…

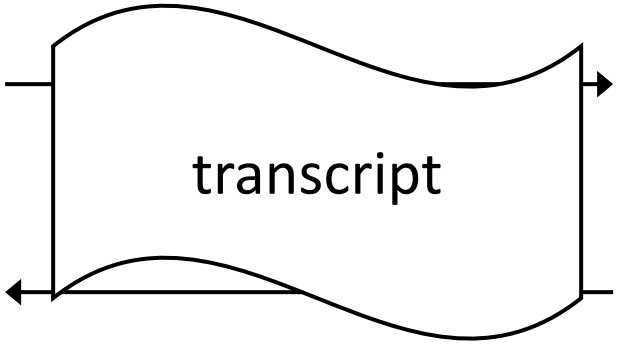
- Main ideas:
  - Some problems exhibit asymmetry – easy to compute, but hard to invert (factoring, RSA, group exponentiation, …)
  - Use this asymmetry to enable two parties to agree on a shared secret key via public discussion(!)
    - Key exchange

Key exchange



Secure against an eavesdropper who sees everything!

More formally…



$k \in \{0,1\}^n$ $k \in \{0,1\}^n$

Security goal: even after observing the transcript, the shared key k should be indistinguishable from a uniform key

Formally

- Fix a key-exchange protocol Π and an attacker (passive eavesdropper) A
- Define the following experiment KEA, $\Pi(n)$:
  - Honest parties run Π using security parameter n, resulting in a transcript trans and (shared) key k
  - Choose uniform bit b. If b=0, then set k'=k; if b=1, then choose uniform $k' \in \{0,1\}n$
  - Give trans and k' to A, which outputs a bit b'
  - Exp't evaluates to 1 (A succeeds) if b'=b

Security

- Key-exchange protocol Π is secure (against passive eavesdropping) if for all probabilistic, poly-time A it holds that

$$\Pr[\text{KEA}, \Pi(n) = 1] \leq \tfrac{1}{2} + \text{negl}(n)$$

Notes

- Being unable to compute the key given the transcript is not a strong enough guarantee

- Indistinguishability of the shared key from uniform is a much stronger guarantee…
  - …and is necessary if the shared key will subsequently be used for private-key crypto!

Diffie-Hellman key exchange

G, q, g, h_1

h_2

$(G, q, g) \leftarrow \mathcal{G}(1^n)$

$x \leftarrow \mathbb{Z}_q$

$h_1 = g^x$

$y \leftarrow \mathbb{Z}_q$

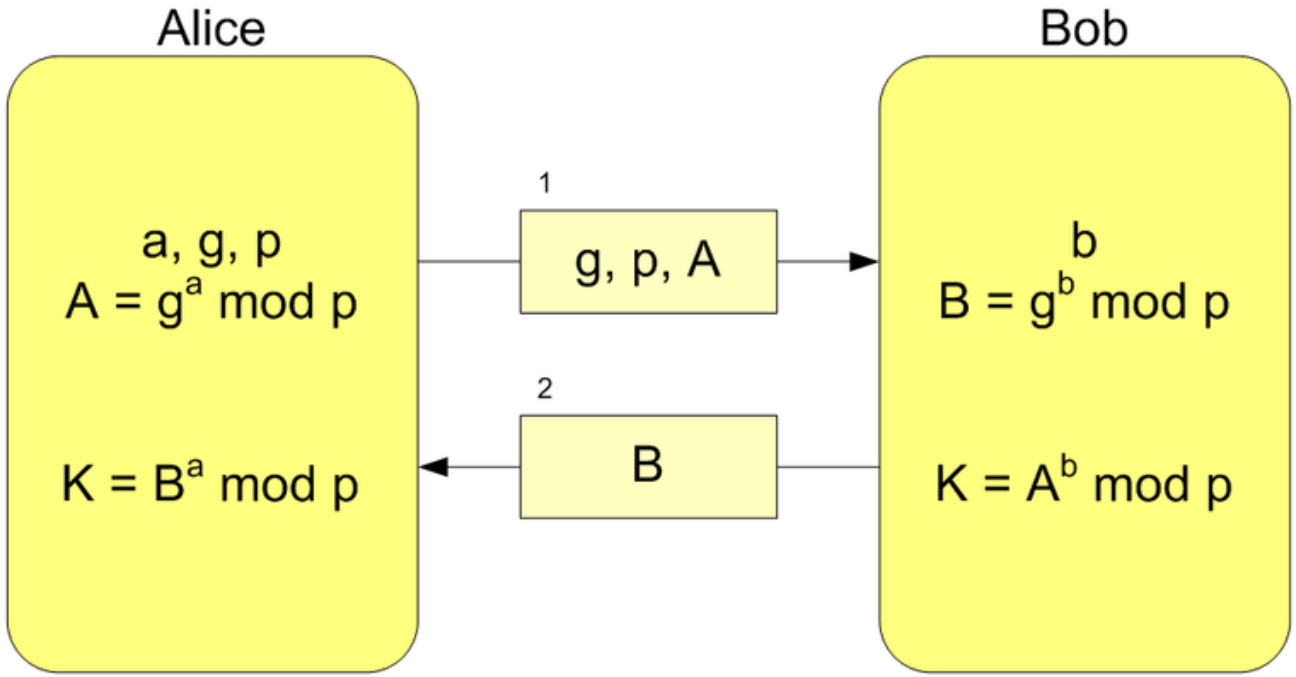
$h_2 = g^y$

$k_1 = (h_2)^x$

$k_2 = (h_1)^y$

Recall…

- Decisional Diffie-Hellman (DDH) assumption:
  - Given G, q, g, gx, gy, cannot distinguish gxy from a uniform group element



$K = A^b \bmod p = (g^a \bmod p)^b \bmod p = g^{ab} \bmod p = (g^b \bmod p)^a \bmod p = B^a \bmod p$

Security?

- Eavesdropper sees G, q, g, gx, gy
- Shared key k is gxy

- Computing k from the transcript is exactly the computational Diffie-Hellman problem

- Distinguishing k from a uniform group element is exactly the decisional Diffie-Hellman problem
  - $\Rightarrow$ If the DDH problem is hard relative to G, this is a secure key-exchange protocol!

Example

- Work in order-11 subgroup of $\mathbb{Z}^*23$
  - Note: 23 and 11 both prime
  - 23 = 2*11 + 1
  - Let G = {x2 | x $\in$ $\mathbb{Z}^*23$}
    - How can you find a generator?

A subtlety

- We want our key-exchange protocol to give us a uniform(-looking) key $k \in \{0,1\}n$
- Instead we have a uniform(-looking) group element $k \in G$
  - Not clear how to use this as, e.g., an AES key

- Solution: key derivation
  - Set k' = H(k) for suitable hash function H
    - Secure if H is modeled as a random oracle

Modern key-exchange protocols

- Security against passive eavesdroppers is insufficient
- Generally want authenticated key exchange
  - This requires some form of setup in advance

- Modern key-exchange protocols provide this
  - We will return to this later