Homework 2

- Due: Tuesday, October 2nd at 3PM (beginning of class)
- Please Typeset Your Solutions (LaTeX, Word etc…)
- You may collaborate, but must write up your own solutions in your own words

Merkle-Damgård Transform

Construction: (Gen,h) fixed length hash function from 2n bits to n bits

$H^s(x)$ =

1. Break x into n bit segments $x_1,..,x_d$ (pad last block by 0's)
2. $z_0 = 0^n$ (initialization)
3. For i = 1 to d
   1. $z_i = h^s(z_{i-1} \parallel x_i)$
4. Output $z_{d+1} = h^s(z_d \parallel L)$ where $L := x_{d+1}$ encodes $|x|$ as an n-bit string

Cryptography
CS 555

Week 6:

- Random Oracle Model
- Applications of Hashing
- Stream Ciphers (time permitting)
- Block Ciphers
- Feistel Networks
- DES, 3DES

Readings: Katz and Lindell Chapter 6-6.2.4

Recap

- Hash Functions
  - Definition
  - Merkle-Damgard
- HMAC construction
- Generic Attacks on Hash Function
  - Birthday Attack
  - Small Space Birthday Attacks (cycle detection)
- Pre-Computation Attacks: Time/Space Tradeoffs

Week 6: Topic 1: Random Oracle Model + Hashing Applications

(Recap) Collision-Resistant Hash Function

Intuition: Hard for computationally bounded attacker to find x,y s.t. H(x) = H(y)

How to formalize this intuition?

- **Attempt 1:** For all PPT A,

$$\Pr\left[A_{x,y}(1^n) = (x,y)\ s.t\ H(x) = H(y)\right] \leq negl(n)$$

- **The Problem:** Let x,y be given s.t. H(x)=H(y)

$$A_{x,y}(1^n) = (x,y)$$

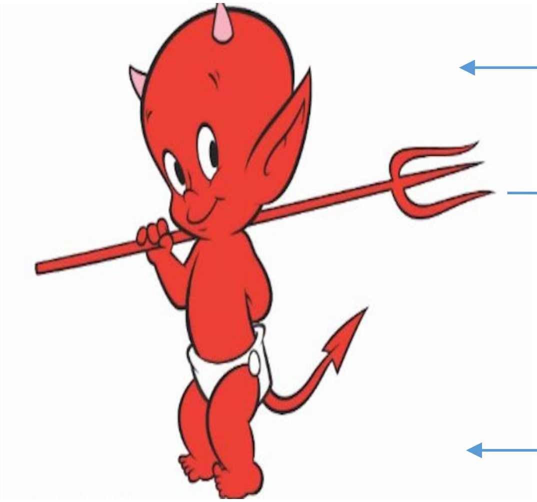
- We are assuming that |x| > |H(x)|. Why?
  - H(x)=x is perfectly collision resistant! (but with no compression)

(Recap) Keyed Hash Function Syntax

- Two Algorithms
  - $\mathrm{Gen}(1^n; R)$ (Key-generation algorithm)
    - Input: Random Bits R
    - Output: ~~Secret~~ key s
  - $H^s(m)$ (Hashing Algorithm)
    - Input: key s and message $\mathrm{m} \in \{0,1\}^*$ (unbounded length)
    - Output: hash value $H^s(m) \in \{0,1\}^{\ell(n)}$

- Fixed length hash function
  - $m \in \{0,1\}^{\ell'(n)}$ with $\ell'(n) > \ell(n)$

Collision Experiment ($HashColl_{A,\Pi}(n)$)



s

x_1,x_2

$$HashColl_{A,\Pi}(n) = \begin{cases} 1 & if\ H^s(x_1) = H^s(x_2) \\ 0 & otherwise \end{cases}$$

$\mathrm{s} = \mathrm{Gen}(1^n; R)$

Definition: (Gen,H) is a collision resistant hash function if

$$\forall PPT\ A\ \exists \mu\ (\text{negligible})\ \text{s.t}$$

$$\Pr\left[HashColl_{A,\Pi}(n)=1\right] \leq \mu(n)$$

When Collision Resistance Isn't Enough

- **Example**: Message Commitment
  - Alice sends Bob: $\mathrm{H}^s(r \parallel m)$ (e.g., predicted winner of NCAA Tournament)
  - Alice can later reveal message (e.g., after the tournament is over)
    - Just send r and m (note: r has fixed length)
    - Why can Alice not change her message?
  - In the meantime Bob shouldn't learn *anything* about m

- **Problem**: Let (Gen,H') be collision resistant then so is (Gen,H)

$$H^s(x_1, \ldots, x_d) = H'^s(x_1, \ldots, x_d) \parallel x_d$$

When Collision Resistance Isn't Enough

- **Problem**: Let (Gen,H') be collision resistant then so is (Gen,H)

$$H^s(x_1, \dots, x_d) = H'^s(x_1, \dots, x_d) \parallel x_d$$

- (Gen,H) definitely does not hide all information about input $(x_1, \dots, x_d)$

- **Conclusion**: Collision resistance is not sufficient for message commitment

The Tension

- **Example**: Message Commitment
  - Alice sends Bob: $\mathrm{H}^s(r \parallel m)$ (e.g., predicted winners of NCAA Final Four)
  - Alice can later reveal message (e.g., after the Final Four is decided)
  - In the meantime Bob shouldn't learn anything about m

This is still a reasonable approach in practice!

- No attacks when instantiated with any reasonable candidate (e.g., SHA3)
- Cryptographic hash functions seem to provide "something" beyond collision resistance, but how do we model this capability?

Random Oracle Model

- Model hash function H as a truly random function
- Algorithms can only interact with H as an oracle
  - **Query**: x
  - **Response**: H(x)
- If we submit the same query you see the same response
- If x has not been queried, then the value of H(x) is uniform

- **Real World:** H instantiated as cryptographic hash function (e.g., SHA3) of fixed length (no Merkle-Damgård)

Back to Message Commitment

- **Example**: Message Commitment
  - Alice sends Bob: $\mathrm{H}(m \parallel r)$ (e.g., predicted winners of NCAA Final Four)
  - Alice can later reveal message (e.g., after the Final Four is decided)
    - Just send r and m (note: r has fixed length)
    - Why can Alice not change her message?
  - In the meantime Bob shouldn't learn *anything* about m

- **Random Oracle Model**: Above message commitment scheme is secure (Alice cannot change m + Bob learns nothing about m)
- **Information Theoretic Guarantee** against any attacker with q queries to H

Random Oracle Model: Pros

- It is easier to prove security in Random Oracle Model

- Suppose we are simulating attacker A in a reduction
  - **Extractability**: When A queries H at x we **see this query** and learn x (and can easily find H(x))
  - **Programmability**: We can set the value of H(x) to a value of our choice
    - As long as the value is correctly distribute i.e., close to uniform
- Both **Extractability** and **Programmability** are useful tools for a security reduction!

Random Oracle Claim

Theorem: Any algorithm A that makes q to a random oracle $H: \{0,1\}^* \to \{0,1\}^n$ will find a collision with probability at most

$$\binom{q}{2} 2^{-n}$$

Proof: For distinct strings x,y we have

$$Pr[H(x) = H(y)] = 2^{-n}.$$

Let $x_1, \ldots, x_q$ denote A's queries to random oracle. By the union bound

$$Pr\left[\exists i < j \leq q \text{ s.t. } H(x_i) = H(x_j)\right] \leq \binom{q}{2} 2^{-n}.$$

Key Derivation

- Transform (low-entropy) password into high-entropy secret key K

$$\mathrm{KDF}(\mathrm{pwd}) = \mathrm{H}(\mathrm{pwd})$$

Suppose that $\mathrm{pwd} \in \{1, \ldots, n\}$ and attacker can make at most $q(n) = \sqrt{n}$ queries to random oracle H.

If attacker does not query $\mathrm{H}(\mathrm{pwd})$ then the secret key $\mathrm{K} = \mathrm{H}(\mathrm{pwd})$ can be viewed as a uniformly random λ-bit string!

→ Probability of violating MAC security with K is at most $\frac{q(n)}{n} + \mathrm{negl}(\lambda)$

Random Oracle Model: Pros

- It is easier to prove security in Random Oracle Model

- Provably secure constructions in random oracle model are often much more efficient (compared to provably secure construction is “standard model”

- Sometimes we only know how to design provably secure protocol in random oracle model

Random Oracle Model: Cons

- Lack of formal justification
- Why should security guarantees translate when we instantiate random oracle with a real cryptographic hash function?

- We can construct (contrived) examples of protocols which are
  - Secure in random oracle model…
  - But broken in the real world

Random Oracle Model: Justification

"A proof of security in the random-oracle model is significantly better than no proof at all."

- **Evidence of sound design** (any weakness involves the hash function used to instantiate the random oracle)
- **Empirical Evidence for Security**

"there have been no successful real-world attacks on schemes proven secure in the random oracle model"

Hash Function Application: Fingerprinting

- The hash h(x) of a file x is a unique identifier for the file
  - Collision Resistance → No need to worry about another file y with H(y)=H(y)

- Application 1: Virus Fingerprinting

- Application 2: P2P File Sharing

- Application 3: Data deduplication

Tamper Resistant Storage

Dropbox

m_1 →

$H(m_1)$

m_1' ←

Tamper Resistant Storage

File Index	Hash
1	$H(m_1)$
2	$H(m_2)$
3	$H(m_3)$

Disadvantage: Too many hashes to store

Dropbox

m_1, m_2, m_3

Send file 1

m_1'

Tamper Resistant Storage

Disadvantage: Need all files to compute hash m_1,m_2,m_3

$H(m_1,m_2,m_3)$

Dropbox

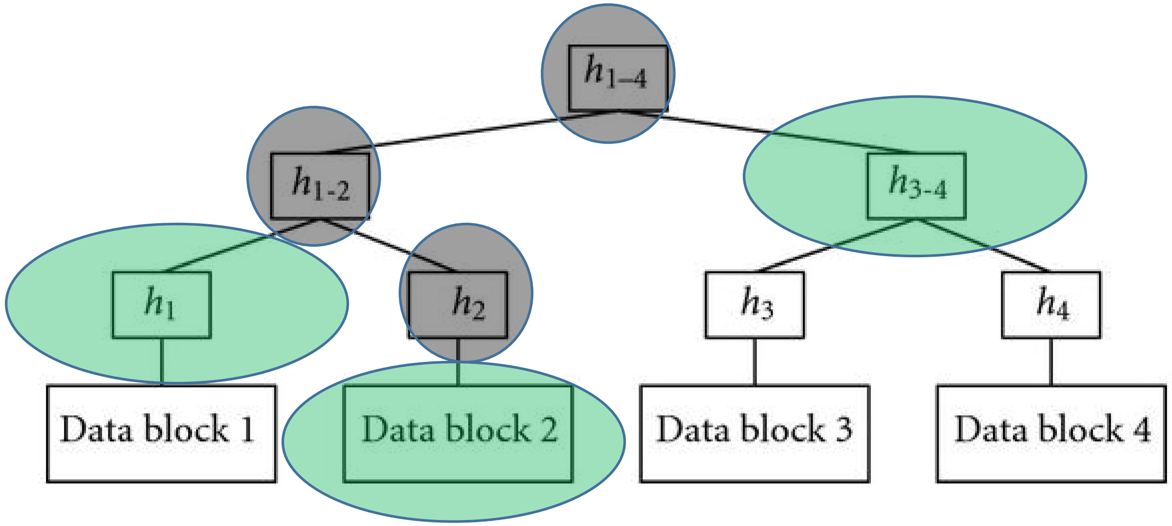
m_1,m_2,m_3

Send file 1

m_1'

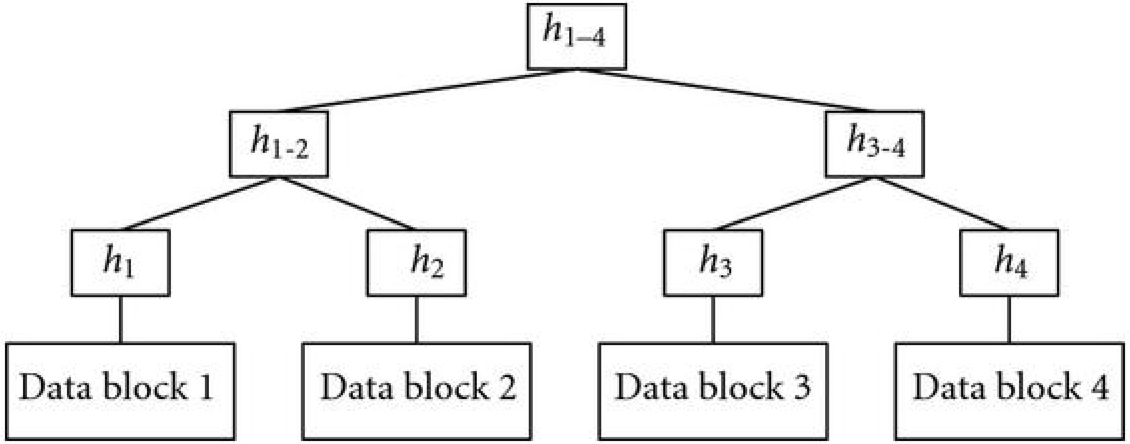
Merkle Trees

- **Proof of Correctness for data block 2**



- **Verify that root matches**
- **Proof consists of just log(n) hashes**
  - **Verifier only needs to permanently store only one hash value**

Merkle Trees



Theorem: Let (Gen, h^s) be a collision resistant hash function and let $H^s(m)$ return the root hash in a Merkle Tree. Then H^s is collision resistant.

Tamper Resistant Storage

Root: $H_{1\text{-}4}$

m_1,m_2,m_3,m_4

Send file 2

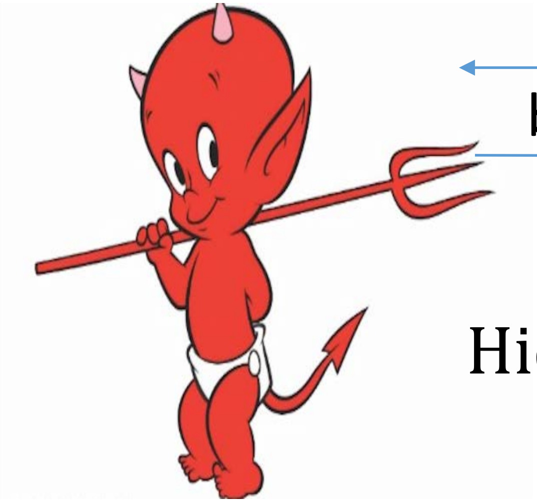
$m_2',h_1,h_{3\text{-}4}$

Dropbox

Commitment Schemes

- Alice wants to commit a message m to Bob
  - And possibly reveal it later at a time of her choosing
- Properties
  - Hiding: commitment reveals nothing about m to Bob
  - Binding: it is infeasible for Alice to alter message

Commitment Hiding $(\text{Hiding}_{A,Com}(n))$



m_0, m_1 →

← **commit(r,m_b)**

b' →

$$\text{Hiding}_{A,Com}(n) = \begin{cases} 1 & \text{if } b = b' \\ 0 & otherwise \end{cases}$$

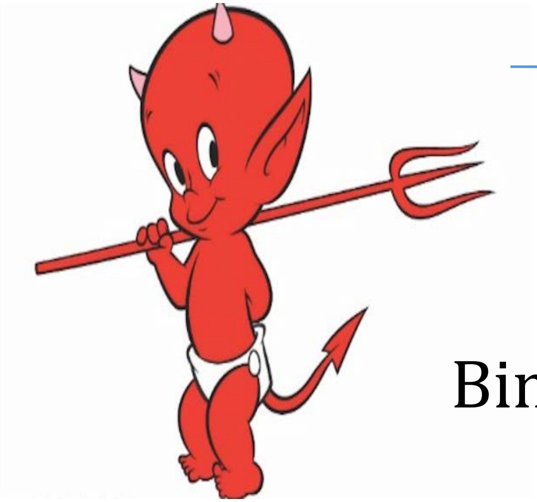
r = Gen(.)

Bit b

$$\forall PPT\ A\ \exists\mu\ (\text{negligible})\ \text{s.t}$$

$$\Pr\left[\text{Hiding}_{A,Com}(n) = 1\right] \leq \frac{1}{2} + \mu(n)$$

Commitment Binding ($\mathrm{Binding}_{A,Com}(n)$)



r_0, r_1, m_0, m_1

$$\mathrm{Binding}_{A,Com}(n) = \begin{cases} 1 & \text{if } \mathbf{commit(r_0,m_0) = commit(r_1,m_1)} \\ 0 & otherwise \end{cases}$$

$$\forall PPT\ A\ \exists \mu\ (\text{negligible})\ \mathrm{s.t}$$
$$\Pr\left[\mathrm{Binding}_{A,Com}(n) = 1\right] \le \mu(n)$$

Secure Commitment Scheme

- **Definition:** A secure commitment scheme is **hiding** and **binding**
- **Hiding**

$$\forall PPT\ A\ \exists\mu\ (\text{negligible})\ \text{s.t}$$

$$\Pr\left[\text{Hiding}_{A,Com}(n) = 1\right] \leq \frac{1}{2} + \mu(n)$$

- **Binding**

$$\forall PPT\ A\ \exists\mu\ (\text{negligible})\ \text{s.t}$$

$$\Pr\left[\text{Binding}_{A,Com}(n) = 1\right] \leq \mu(n)$$

Commitment Scheme in Random Oracle Model

- $\mathbf{Commit}(\mathrm{r}, \mathrm{m}) \coloneqq \mathrm{H}(r \parallel m)$

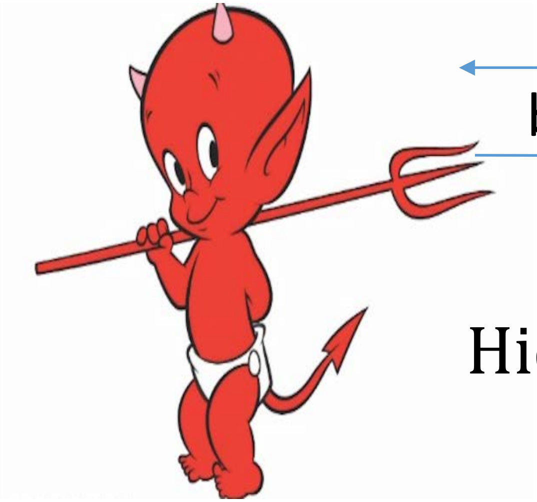
- $\mathbf{Reveal}(\mathrm{c}) \coloneqq (\mathrm{r}, \mathrm{m})$

Theorem: In the random oracle model this is a secure commitment scheme.

Binding:

$$\mathbf{commit}(\mathbf{r}_0,\mathbf{m}_0)\mathbf{=}\ \mathbf{commit}(\mathbf{r}_1,\mathbf{m}_1) \leftrightarrow \mathrm{H}(\mathbf{r}_0 \parallel \mathbf{m}_0)=\mathrm{H}(\mathbf{r}_1 \parallel \mathbf{m}_1)$$

Commitment Hiding $(\mathrm{Hiding}_{A,Com}(n))$



m_0, m_1 →

← $\mathrm{H}(r \parallel m_b)$

b' →

$$\mathrm{Hiding}_{A,Com}(n) = \begin{cases} 1 & if\, b = b' \\ 0 & otherwise \end{cases}$$

r = Gen(.)

Bit b

$$\forall \cancel{PPT}\ A\ making\ q(n)\ queries\ \mathrm{s.t}$$

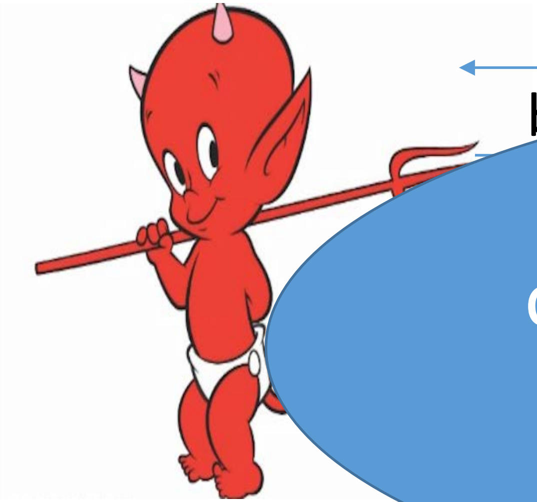
$$\Pr\left[\mathrm{Hiding}_{A,Com}(n) = 1\right] \le \frac{1}{2} + \frac{q(n)}{2^{|r|}}$$

Commitment Hiding $\left(\mathrm{Hiding}_{A,Com}(n)\right)$

m_0, m_1

$\mathrm{H}(r \parallel m_b)$

b’



If attacker never makes query of the form $\mathrm{H}(r \parallel x)$ then bit b is information theoretically hidden

r = Gen(.)

Bit b

$\forall \cancel{PPT}\ A\ making\ q(n)\ quer$ s.t

$$\Pr\left[\mathrm{Hiding}_{A,Com}(n) = 1\right] \le \frac{1}{2} + \frac{q(n)}{2^{|r|}}$$

Other Applications

- Password Hashing
- Key Derivation
- Later
  - Key Encapsulation Mechanism
  - RSA-FDH etc...

CS 555: Week 6: Topic 6
Block Ciphers

An Existential Crisis?

- We have used primitives like PRGs, PRFs to build secure MACs, CCA-Secure Encryption, Authenticated Encryption etc…

- Do such primitives exist in practice?

- How do we build them?

Recap

- Hash Functions/PRGs/PRFs, CCA-Secure Encryption, MACs

Goals for This Week:

- Practical Constructions of Symmetric Key Primitives

Today's Goals: Block Ciphers

- Sbox
- Confusion Diffusion Paradigm
- Feistel Networks

Pseudorandom Permutation

A keyed function $\mathrm{F}: \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$, which is invertible and "looks random" without the secret key k.

- Similar to a PRF, but
- Computing $F_k(x)$ *and* $F_k^{-1}(x)$ is efficient (polynomial-time)

Definition 3.28: A keyed function $\mathrm{F}: \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ is a **strong pseudorandom permutation** if for all PPT distinguishers D there is a negligible function μ s.t.

$$\left|Pr\left[D^{F_k(.),F_k^{-1}(.)}(1^n)\right] - Pr\left[D^{f(.),f^{-1}(.)}(1^n)\right]\right| \leq \mu(n)$$

Pseudorandom Permutation

Definition 3.28: A keyed function $\mathrm{F}: \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$ is a **strong pseudorandom permutation** if for all PPT distinguishers D there is a negligible function μ s.t.

$$\left| Pr\left[D^{\boxed{F_k(.), F_k^{-1}(.)}}(1^n) \right] - Pr\left[D^{\boxed{f(.), f^{-1}(.)}}(1^n) \right] \right| \leq \mu(n)$$

Notes:

- the first probability is taken over the uniform choice of $k \in \{0,1\}^n$ as well as the randomness of D.
- the second probability is taken over uniform choice of f ∈$\mathbf{Perm_n}$as well as the randomness of D.
- D is *never* given the secret k
- However, D is given oracle access to keyed permutation and inverse

How many permutations?

- **$|Perm_n|$=?**
- **Answer:** $2^n!$
- How many bits to store f ∈**$Perm_n$**?
- **Answer**:

$$\log(2^n!) = \sum_{i=1}^{2^n} \log(i)$$

$$\geq \sum_{i=2^{n-1}}^{2^n} n - 1 \geq (n-1) \times 2^{n-1}$$

How many bits to store permutations?

$$\log(2^n!) = \sum_{i=1}^{2^n} \log(i)$$

$$\geq \sum_{i=2^{n-1}}^{2^n} n - 1 \geq (n-1) \times 2^{n-1}$$

Example: Storing f $\in$ $\mathbf{Perm_{50}}$ requires over 6.8 petabytes (10^{15})

Example 2: Storing f $\in$ $\mathbf{Perm_{100}}$ requires about 12 yottabytes (10^{24})

Example 3: Storing f $\in$ $\mathbf{Perm_{8}}$ requires about 211 bytes

Attempt 1: Pseudorandom Permutation

- Select 16 random permutations on 8-bits $f_1,\ldots,f_{16} \in \mathbf{Perm_8}$.

- **Secret key:** $\mathrm{k} = f_1,\ldots,f_{16}$ (about 3 KB)
- **Input**: $x=x_1,\ldots,x_{16}$ (16 bytes)

$$\mathrm{F_k}(x) = f_1(x_1) \parallel f_2(x_2) \parallel \cdots \parallel f_{16}(x_{16})$$

- Any concerns?

Attempt 1: Pseudorandom Permutation

- Select 16 random permutations on 8-bits $f_1,\ldots,f_{16} \in \mathbf{Perm_8}$.

$$\mathrm{F_k}(x) = \mathrm{f_1(x_1)} \parallel \mathrm{f_2(x_2)} \parallel \cdots \parallel \mathrm{f_{16}(x_{16})}$$

- Any concerns?

$$\mathrm{F_k}(\mathrm{x_1} \parallel \mathrm{x_2} \parallel \cdots \parallel \mathrm{x_{16}}) = \mathrm{f_1(x_1)} \parallel \mathrm{f_2(x_2)} \parallel \cdots \parallel \mathrm{f_{16}(x_{16})}$$

$$\mathrm{F_k}(\mathbf{0} \parallel \mathrm{x_2} \parallel \cdots \parallel \mathrm{x_{16}}) = \mathbf{f_1(0)} \parallel \mathrm{f_2(x_2)} \parallel \cdots \parallel \mathrm{f_{16}(x_{16})}$$

- Changing a bit of input produces insubstantial changes in the output.
- A truly random permutation $\mathrm{F} \in \mathbf{Perm_{128}}$ would not behave this way!

Pseudorandom Permutation Requirements

- Consider a truly random permutation $\mathrm{F} \in \mathbf{Perm}_{\mathbf{128}}$
- Let inputs x and x' differ on a single bit
- We expect outputs F(x) and F(x') to differ on approximately half of their bits
  - F(x) and F(x') should be (essentially) independent.
- A pseudorandom permutation must exhibit the same behavior!

Confusion-Diffusion Paradigm

- Our previous construction was not pseudorandom, but apply the permutations do accomplish something
  - They introduce confusion into F
  - Attacker cannot invert (after seeing a few outputs)
- Approach:
  - **Confuse**: Apply random permutations f_1,…, to each block of input to obtain y_1,…,
  - **Diffuse**: Mix the bytes y_1,…, to obtain byes z_1,…,
  - **Confuse**: Apply random permutations f_1,…, with inputs z_1,…,
  - Repeat as necessary

Attempt 1: Pseudorandom Permutation

- Select 16 random permutations on 8-bits $f_1,\ldots,f_{16} \in \mathbf{Perm_8}$.

$$\mathrm{F_k}(x) = \mathrm{f_1(x_1)} \parallel \mathrm{f_2(x_2)} \parallel \cdots \parallel \mathrm{f_{16}(x_{16})}$$

- Any concerns?

$$\mathrm{F_k}(\mathrm{x_1} \parallel \mathrm{x_2} \parallel \cdots \parallel \mathrm{x_{16}}) = \mathrm{f_1(x_1)} \parallel \mathrm{f_2(x_2)} \parallel \cdots \parallel \mathrm{f_{16}(x_{16})}$$

$$\mathrm{F_k}(\mathbf{0} \parallel \mathrm{x_2} \parallel \cdots \parallel \mathrm{x_{16}}) = \mathbf{f_1(0)} \parallel \mathrm{f_2(x_2)} \parallel \cdots \parallel \mathrm{f_{16}(x_{16})}$$

- Changing a bit of input produces insubstantial changes in the output.
- A truly random permutation $\mathrm{F} \in \mathbf{Perm_{128}}$ would not behave this way!

Confusion-Diffusion Paradigm

Example:

- Select 8 random permutations on 8-bits $f_1,\ldots,f_{16} \in \mathbf{Perm_8}$
- Select 8 extra random permutations on 8-bits $g_1,\ldots,g_8 \in \mathbf{Perm_8}$

$\mathrm{F_k}(x_1 \parallel x_2 \parallel \cdots \parallel x_8)$=

1. $y_1 \parallel \cdots \parallel y_8 := f_1(x_1) \parallel f_2(x_2) \parallel \cdots \parallel f_8(x_8)$
2. $z_1 \parallel \cdots \parallel z_8 := \mathbf{Mix}(y_1 \parallel \cdots \parallel y_8)$
3. **Output:** $f_1(z_1) \parallel f_2(z_2) \parallel \cdots \parallel f_8(z_8)$

Example Mixing Function

Mix$(y_1 \parallel \cdots \parallel y_8)$=

1. For i=1 to 8
2. $\qquad z_i := y_1[i] \parallel \cdots \parallel y_8[i]$
3. End For
4. **Output:** $g_1(z_1) \parallel g_2(z_2) \parallel \cdots \parallel g_8(z_8)$

$$\begin{array}{rc} & \begin{array}{ccc} z_1 & & z_8 \end{array} \\ \begin{array}{r} y_1 = \\ \\ y_8 = \end{array} & \begin{bmatrix} y_1[1] & \cdots & y_1[8] \\ \vdots & \ddots & \vdots \\ y_8[1] & \cdots & y_8[8] \end{bmatrix} \end{array}$$

Are We Done?

$F_k(x_1 \parallel x_2 \parallel \cdots \parallel x_8)=$

1. $y_1 \parallel \cdots \parallel y_8 := f_1(x_1) \parallel f_2(x_2) \parallel \cdots \parallel f_8(x_8)$
2. $z_1 \parallel \cdots \parallel z_8 := \mathbf{Mix}(y_1 \parallel \cdots \parallel y_8)$
3. **Output:** $f_1(z_1) \parallel f_2(z_2) \parallel \cdots \parallel f_8(z_8)$

$$\begin{array}{cc} & \begin{array}{ccc} z_1 & \quad & z_8 \end{array} \\ \begin{array}{c} y_1 = \\ \\ y_8 = \end{array} & \begin{bmatrix} y_1[1] & \cdots & y_1[8] \\ \vdots & \ddots & \vdots \\ y_8[1] & \cdots & y_8[8] \end{bmatrix} \end{array}$$

Suppose $f_1(x_1) = 00110101 = y_1$ and $f_1(x'_1) = 00110101 = y'_1$

$F_k(x'_1 \parallel x_2 \parallel \cdots \parallel x_8)=$

1. $y'_1 \parallel \cdots \parallel y_8 := f_1(x'_1) \parallel f_2(x_2) \parallel \cdots \parallel f_8(x_8)$
2. $z_1 \parallel \cdots \parallel z'_8 := \mathbf{Mix}(y'_1 \parallel \cdots \parallel y_8)$
3. **Output:** $f_1(z_1) \parallel f_2(z_2) \parallel \cdots \parallel f_8(z'_8)$

Highly unlikely that a truly random permutation would behave this way!

Substitution Permutation Networks

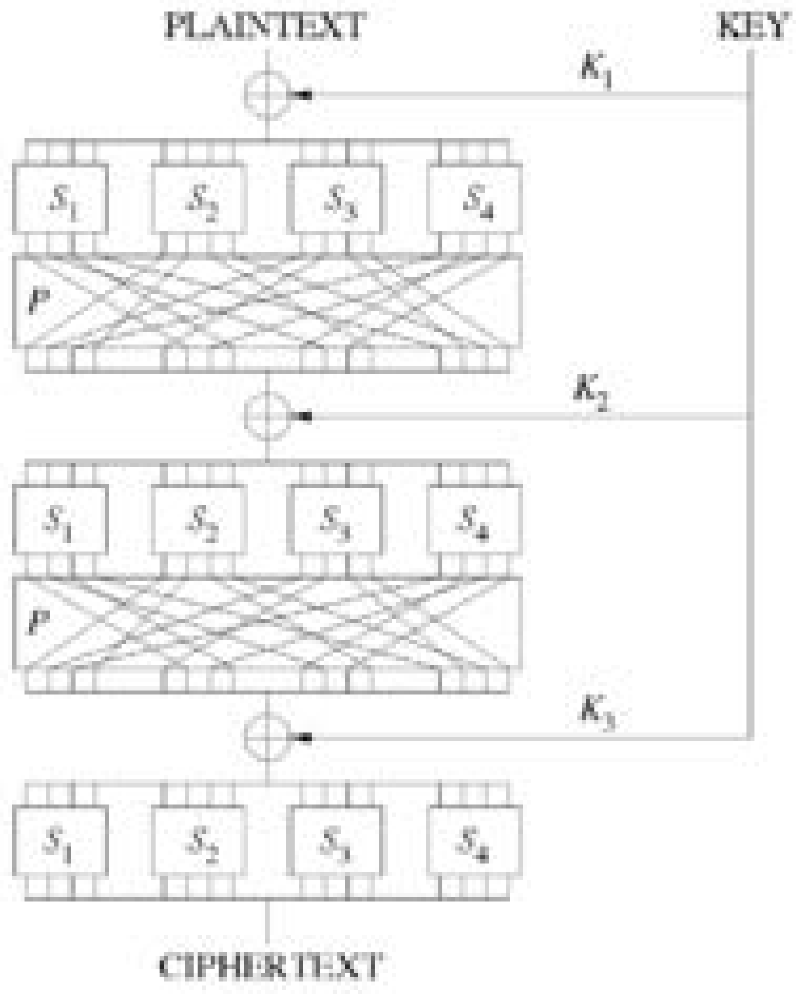
- S-box a public "substitution function" (e.g.$S \in$ **Perm$_8$**).
- S is not part of a secret key, but can be used with one

$$f(x) = S(x \oplus k)$$

- Input to round: x, k (k is subkey for current round)
- **Key Mixing**: Set $x \coloneqq x \oplus k$
- **Substitution**: $x \coloneqq S_1(x_1) \parallel S_2(x_2) \parallel \cdots \parallel S_8(x_8)$
- **Bit Mixing Permutation**: permute the bits of x to obtain the round output

Note: there are only n! possible bit mixing permutations of [n] as opposed to 2^n! Permutations of $\{0,1\}^n$

Substitution Permutation Networks



- **Proposition 6.3:** Let F be a keyed function defined by a Substitution Permutation Network. Then for any keys/number of rounds F_k is a permutation.

- Why? Composing permutations f,g results in another permutation h(x)=g(f(x)).

Remarks

- Want to achieve “avalanche effect” (one bit change should “affect” every output bit)
- Should a S-box be a random byte permutation?
- Better to ensure that S(x) differs from x on at least 2-bits (for all x)
  - Helps to maximize “avalanche effect”
- Mixing Permutation should ensure that output bits of any given S-box are used as input to multiple S-boxes in the next round

Remarks

- How many rounds?

- **Informal Argument:** If we ensure that S(x) differs from S(x') on at least 2-bits (for all x,x' differing on at least 1 bit) then every input bit affects
  - 2 bits of round 1 output
  - 4 bits of round 2 output
  - 8 bits of round 3 output
  - ….
  - 128 bits of round 4 output

- Need at least 7 rounds (minimum) to ensure that every input bit affects every output bit

Attacking Lower Round SPNs

- Trivial Case: One full round with no final key mixing step
- **Key Mixing**: Set $\mathrm{x} \coloneqq \mathrm{x} \oplus k$
- **Substitution**: $\mathrm{y} \coloneqq \mathrm{S}_1(\mathrm{x}_1) \parallel \mathrm{S}_2(\mathrm{x}_2) \parallel \cdots \parallel \mathrm{S}_8(\mathrm{x}_8)$
- **Bit Mixing Permutation**: P permute the bits of y to obtain the round output

- Given input/output $(x, F_k(x))$
  - Permutations P and S_i are public and can be run in reverse
  - $P^{-1}(F_k(x)) = \mathrm{S}_1(\mathrm{x}_1 \oplus k_1) \parallel \mathrm{S}_2(\mathrm{x}_2 \oplus k_2) \parallel \cdots \parallel \mathrm{S}_8(\mathrm{x}_8 \oplus k_8)$
  - $x_i \otimes k_i = S_i^{-1}\left(\mathrm{S}_1(\mathrm{x}_1 \oplus k_1)\right)$
  - Attacker knows x_i and can thus obtain k_i

Attacking Lower Round SPNs

- Easy Case: One full round with final key mixing step
- **Key Mixing**: Set $\mathrm{x} := \mathrm{x} \otimes k_1$
- **Substitution**: $\mathrm{y} := \mathrm{S}_1(\mathrm{x}_1) \parallel \mathrm{S}_2(\mathrm{x}_2) \parallel \cdots \parallel \mathrm{S}_8(\mathrm{x}_8)$
- **Bit Mixing Permutation**: $\mathrm{z}_1 \parallel \cdots \parallel \mathrm{z}_8 = \mathrm{P}(\mathrm{y})$
- **Final Key Mixing**: Output $\mathrm{z} \oplus k_2$

- Given input/output $(x, F_k(x))$
  - Permutations P and S_i are public and can be run in reverse once k_2 is known
  - Immediately yields attack in 2^{64} time (k_1,k_2 are each 64 bit keys) which narrows down key-space to 2^{64} but we can do much better!

Attacking Lower Round SPNs

- Easy Case: One full round with final key mixing step
- **Key Mixing**: Set $\mathrm{x} \coloneqq \mathrm{x} \oplus k_1$
- **Substitution**: $\mathrm{y} \coloneqq \mathrm{S}_1(\mathrm{x}_1) \parallel \mathrm{S}_2(\mathrm{x}_2) \parallel \cdots \parallel \mathrm{S}_8(\mathrm{x}_8)$
- **Bit Mixing Permutation**: $\mathrm{z}_1 \parallel \cdots \parallel \mathrm{z}_8$ =P(y)
- **Final Key Mixing**: Output $\mathrm{z} \oplus k_2$

- Given input/output $(x, F_k(x))$
  - Permutations P and S_i are public and can be run in reverse once k_2 is known
  - Guessing 8 specific bits of k_2 (which bits depends on P) we can obtain one value $\mathrm{y_i} = \mathrm{S_i}(\mathrm{x_i} \otimes k_\mathrm{i})$
  - Attacker knows x_i and can thus obtain k_i by inverting S_i and using XOR
  - Narrows down key-space to 2^{64} , but in time $8 \mathrm{x} 2^8$

Attacking Lower Round SPNs

- Easy Case: One full round with final key mixing step
- **Key Mixing**: Set $\mathrm{x} \coloneqq \mathrm{x} \oplus k_1$
- **Substitution**: $\mathrm{y} \coloneqq \mathrm{S}_1(\mathrm{x}_1) \parallel \mathrm{S}_2(\mathrm{x}_2) \parallel \cdots \parallel \mathrm{S}_8(\mathrm{x}_8)$
- **Bit Mixing Permutation**: $\mathrm{z}_1 \parallel \cdots \parallel \mathrm{z}_8 = \mathrm{P}(\mathrm{y})$
- **Final Key Mixing**: Output $\mathrm{z} \oplus k_2$

- Given several input/output pairs $(x_j, F_k(x_j))$
  - Can quickly recover k_1 and k_2

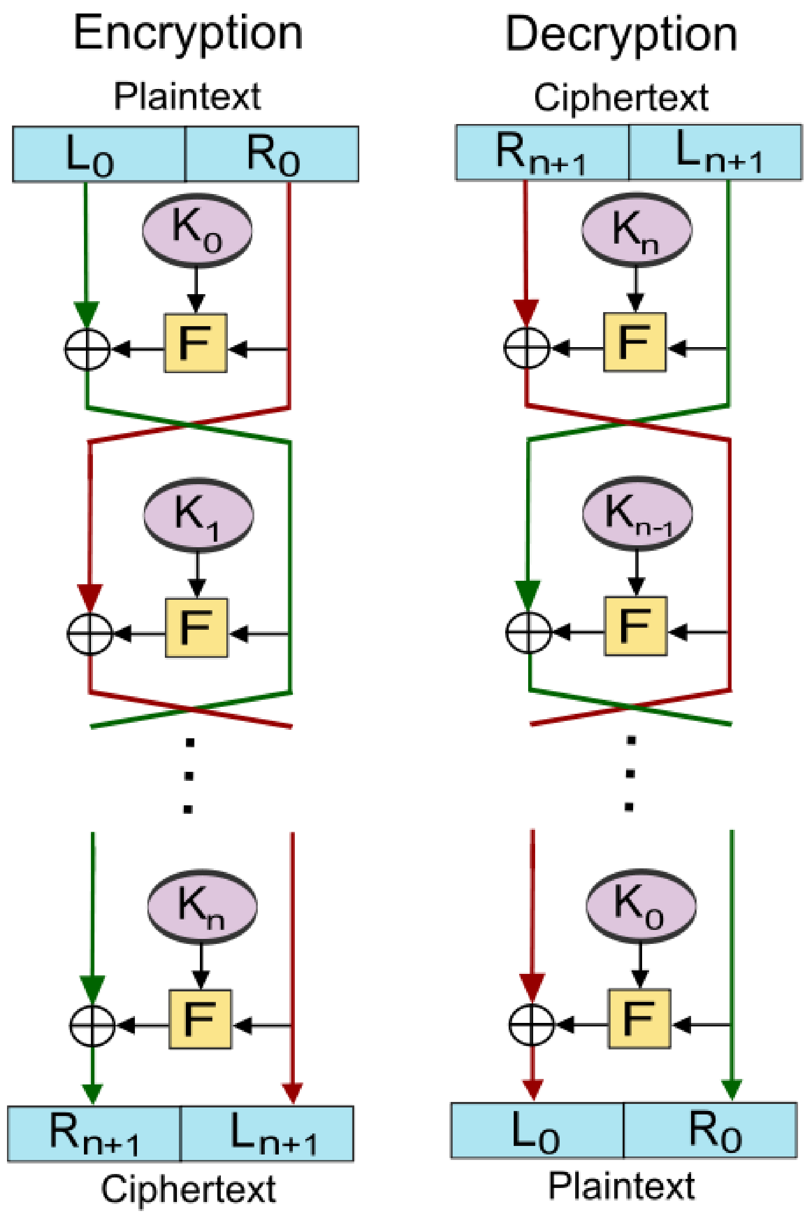
Attacking Lower Round SPNs

- Harder Case: Two round SPN
- Exercise ☺

Feistel Networks

- Alternative to Substitution Permutation Networks

- **Advantage**: underlying functions need not be invertible, but the result is still a permutation



- $R_{i-1} = L_i$
- $L_{i-1} := R_i \oplus F_{k_i}(R_{i-1})$

Proposition: the function is invertible.

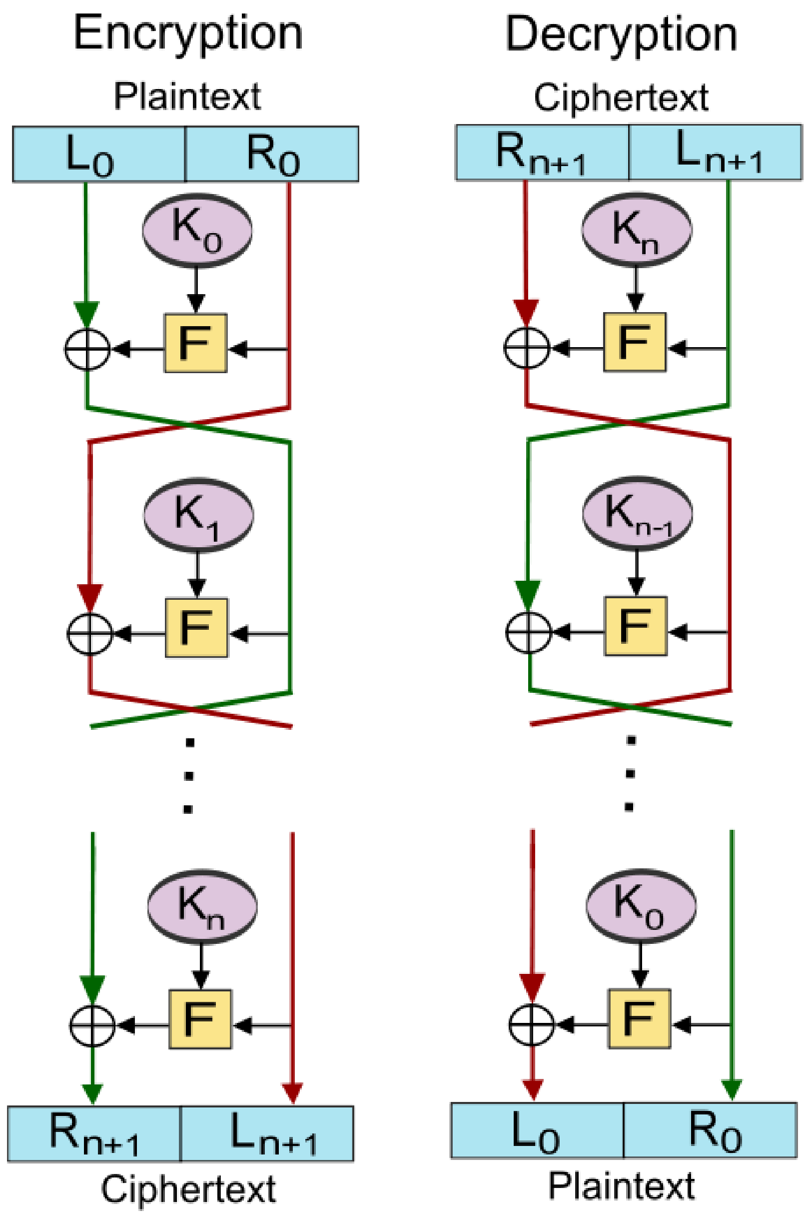
Digital Encryption Standard (DES): 16-round Feistel Network.

CS 555: Week 6: Topic 4
DES, 3DES

Feistel Networks

- Alternative to Substitution Permutation Networks

- **Advantage**: underlying functions need not be invertible, but the result is still a permutation



- $L_{i+1} = R_i$
- $R_{i+1} := L_i \oplus F_{k_i}(R_i)$

Proposition: the function is invertible.

Data Encryption Standard

- Developed in 1970s by IBM (with help from NSA)
- Adopted in 1977 as Federal Information Processing Standard (US)
- Data Encryption Standard (DES): 16-round Feistel Network.
- Key Length: 56 bits
  - Vulnerable to brute-force attacks in modern times
  - 1.5 hours at 14 trillion DES evals/second e.g., Antminer S9 runs at 14 TH/s

DES Round

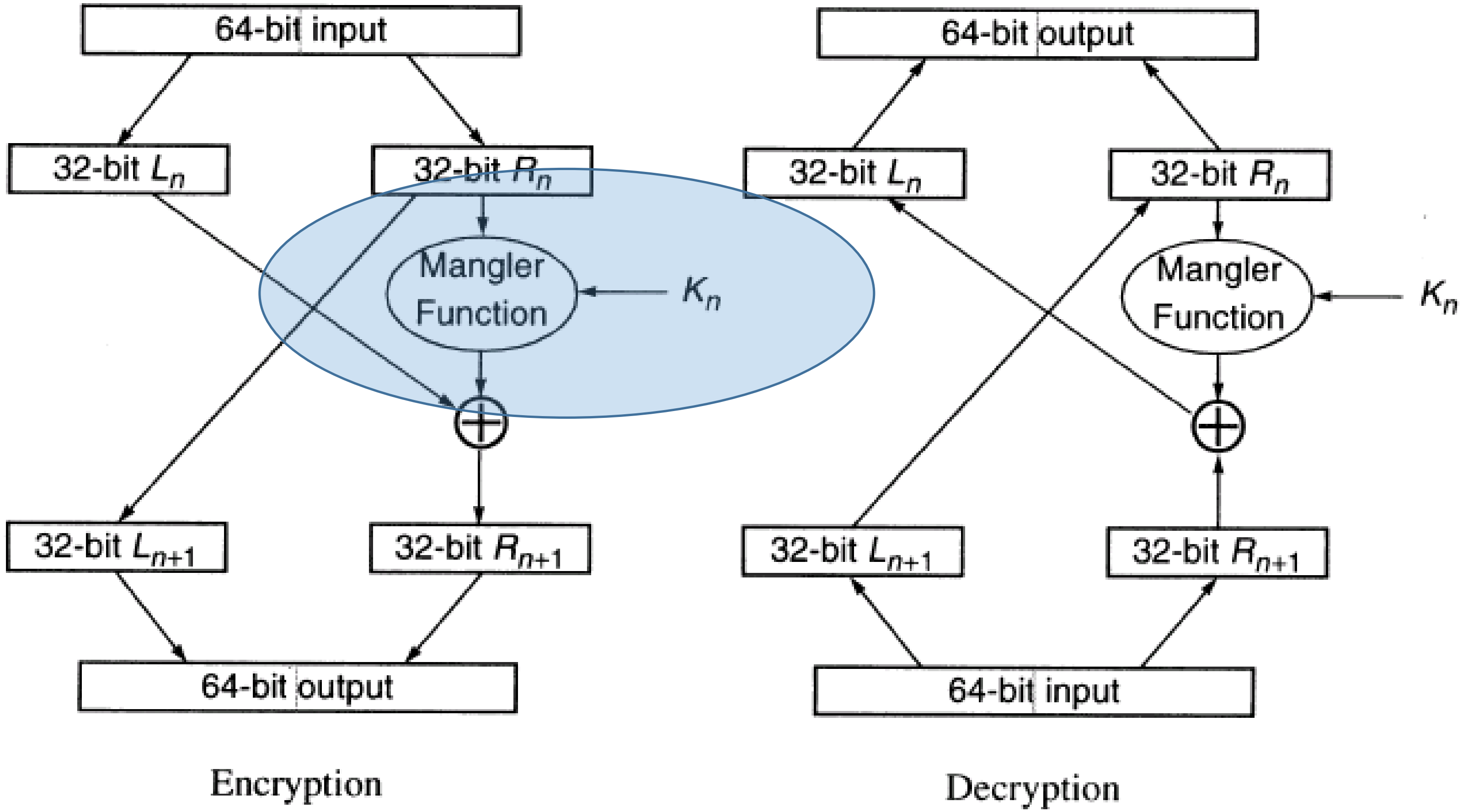
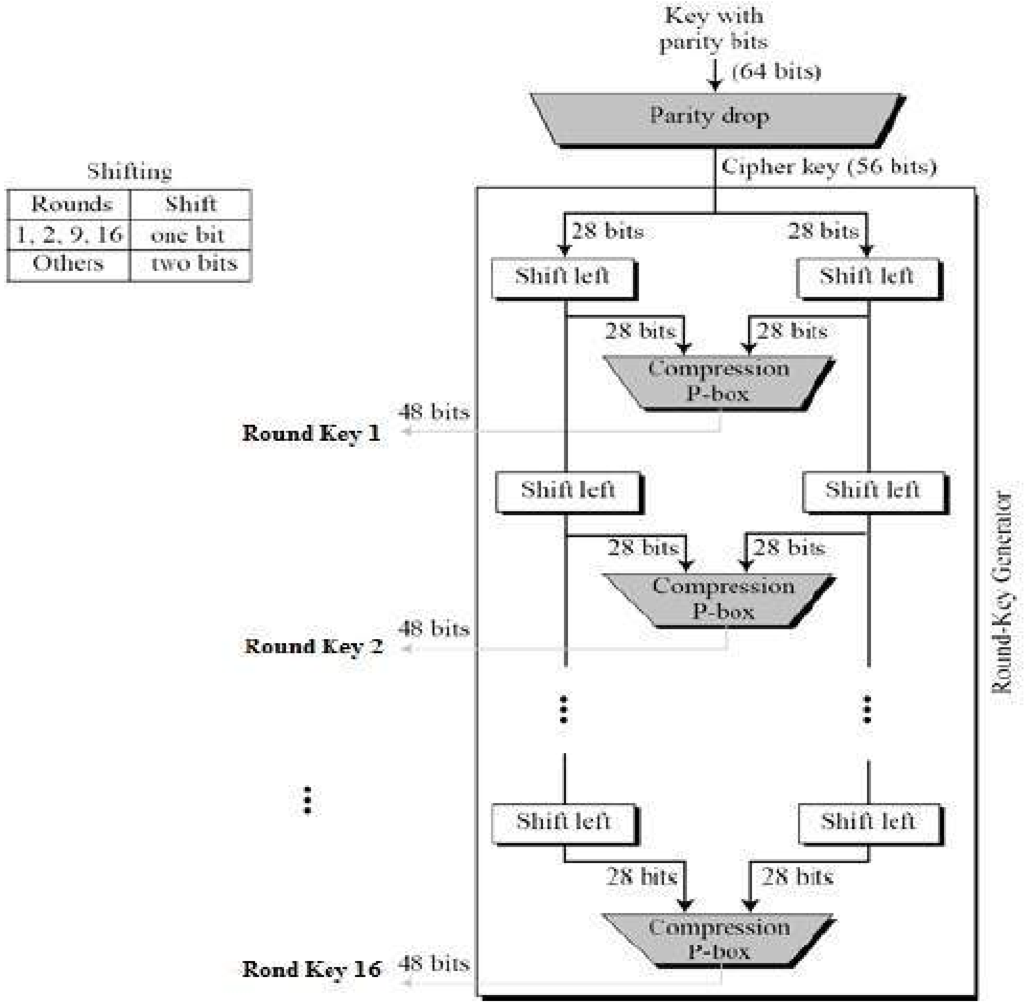


Figure 3-6. DES Round

Generating the Round Keys

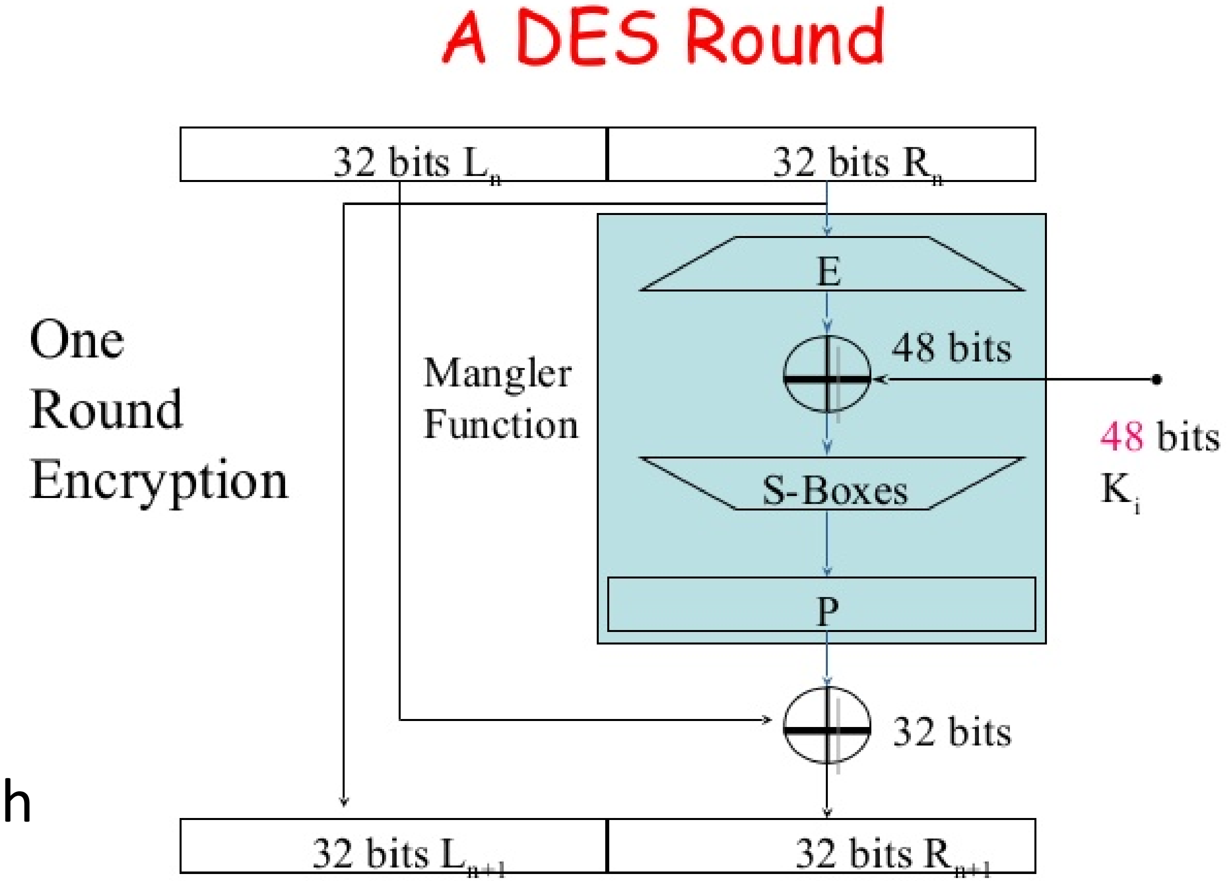
- **Initial Key:** 64 bits
- **Effective Key Length:** 56 bits
- **Round Key Length:** 48 bits (each)

- **16 round keys** derived from initial key

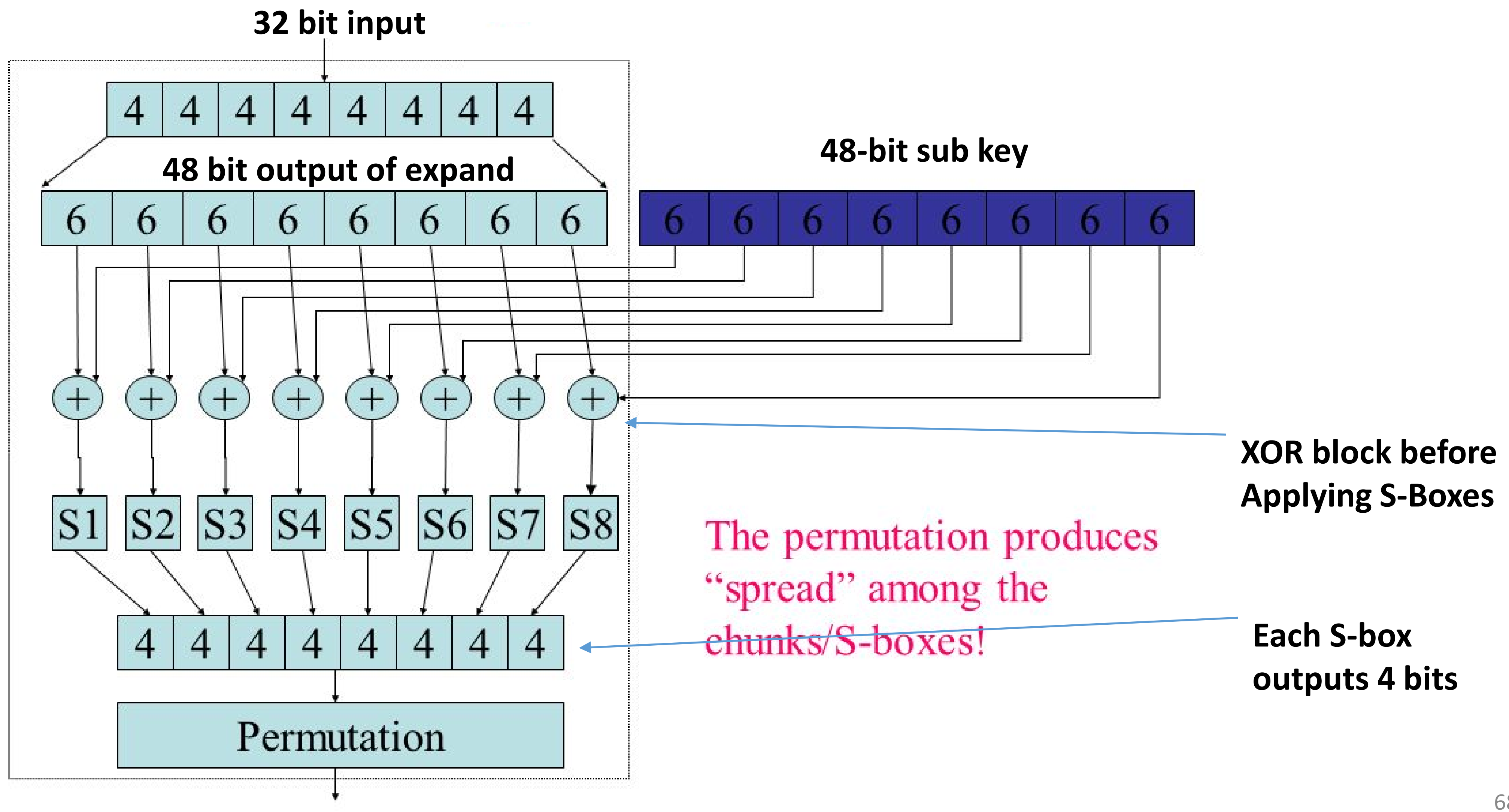


DES Mangle Function

- Expand E: 32-bit input → 48-bit output (duplicates 16 bits)
- S-boxes: $S_1,\ldots,S_8$
  - Input: 6-bits
  - Output: 4 bits
  - Not a permutation!
- 4-to-1 function
  - Exactly four inputs mapped to each possible output



Mangle Function



S-Box Representation as Table

4 columns (2 bits)

16 columns (4 bits)

	00	01	10	11
0000				
0001				
0010				
0011				
0100				
0101				
0110				S(x)=1101
….	….	….	….	….
1111				

x =101101 S(x) = Table[0110,11]

S-Box Representation

Each column is permutation

4 columns (2 bits)

16 columns (4 bits)

	00	01	10	11
0000				
0001				
0010				
0011				
0100				
0101				
0110				S(x)=1101
….	….	….	….	….
1111				

x =101101

S(x) = T[0110,11]

Pseudorandom Permutation Requirements

- Consider a truly random permutation $\mathrm{F} \in$ $\mathbf{Perm_{128}}$
- Let inputs x and x' differ on a single bit
- We expect outputs F(x) and F(x') to differ on approximately half of their bits
  - F(x) and F(x') should be (essentially) independent.
- A pseudorandom permutation must exhibit the same behavior!
- **Requirement**: DES Avalanche Effect!

DES Avalanche Effect

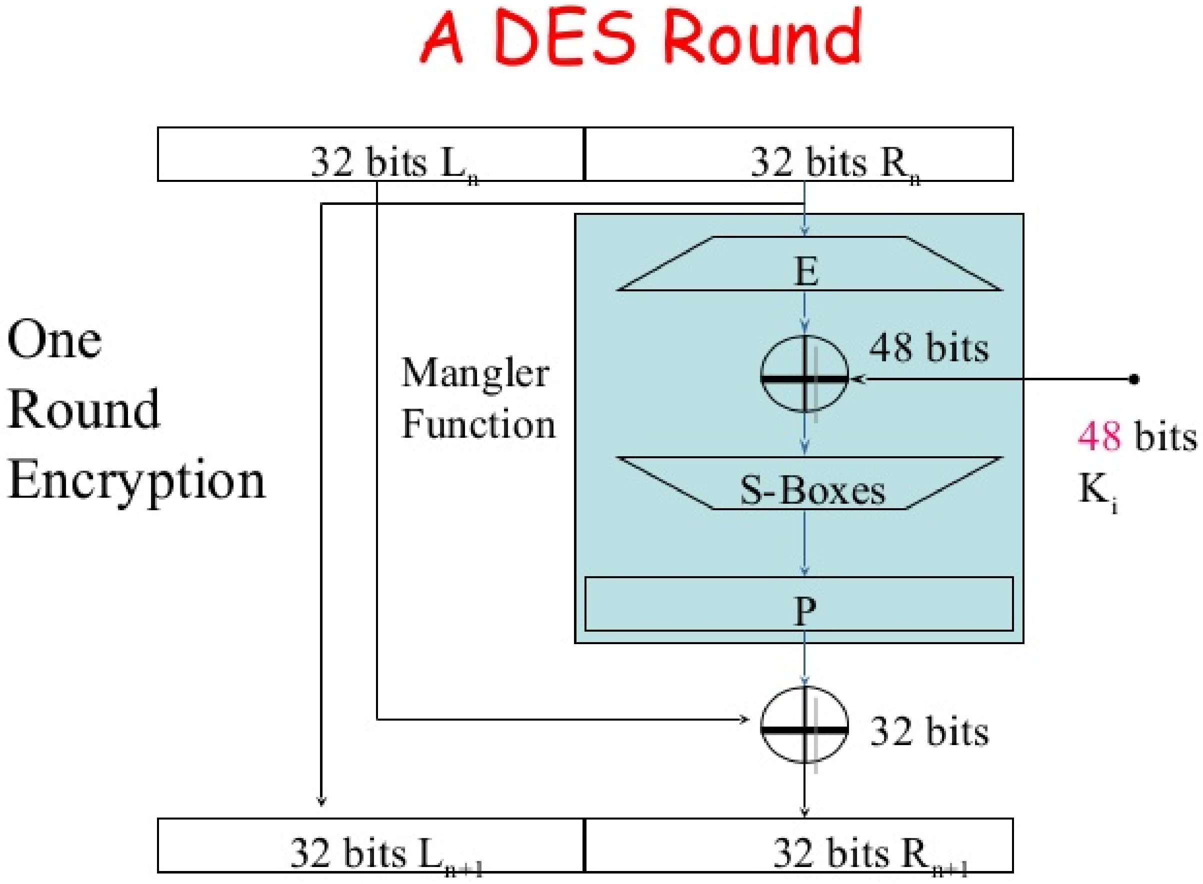
- Permutation the end of the mangle function helps to mix bits

- Special S-box property #1

Let x and x’ differ on one bit then $S_i(x)$ differs from $S_i(x')$ on two bits.

Avalanche Effect Example

- Consider two 64 bit inputs
  - (L_n,R_n) and $(L_n',R'_n=R_n)$
  - L_n and L_n' differ on one bit
- This is worst case example
  - $L_{n+1} = L_{n+1}'=R_n$
  - But now R'_{n+1} and R_{n+1} differ on one bit
- Even if we are unlucky $E(R'_{n+1})$ and $E(R_{n+1})$ differ on 1 bit
- → R_{n+2} and R'_{n+2} differ on two bits
- → $L_{n+2} = R'_{n+1}$ and $L_{n+2}' = R'_{n+1}$ differ in one bit



Avalanche Effect Example

- R_{n+2} and R'_{n+2} differ on two bits
- $L_{n+2} = R_{n+1}$ and $L_{n+2}' = R'_{n+1}$ differ in one bit

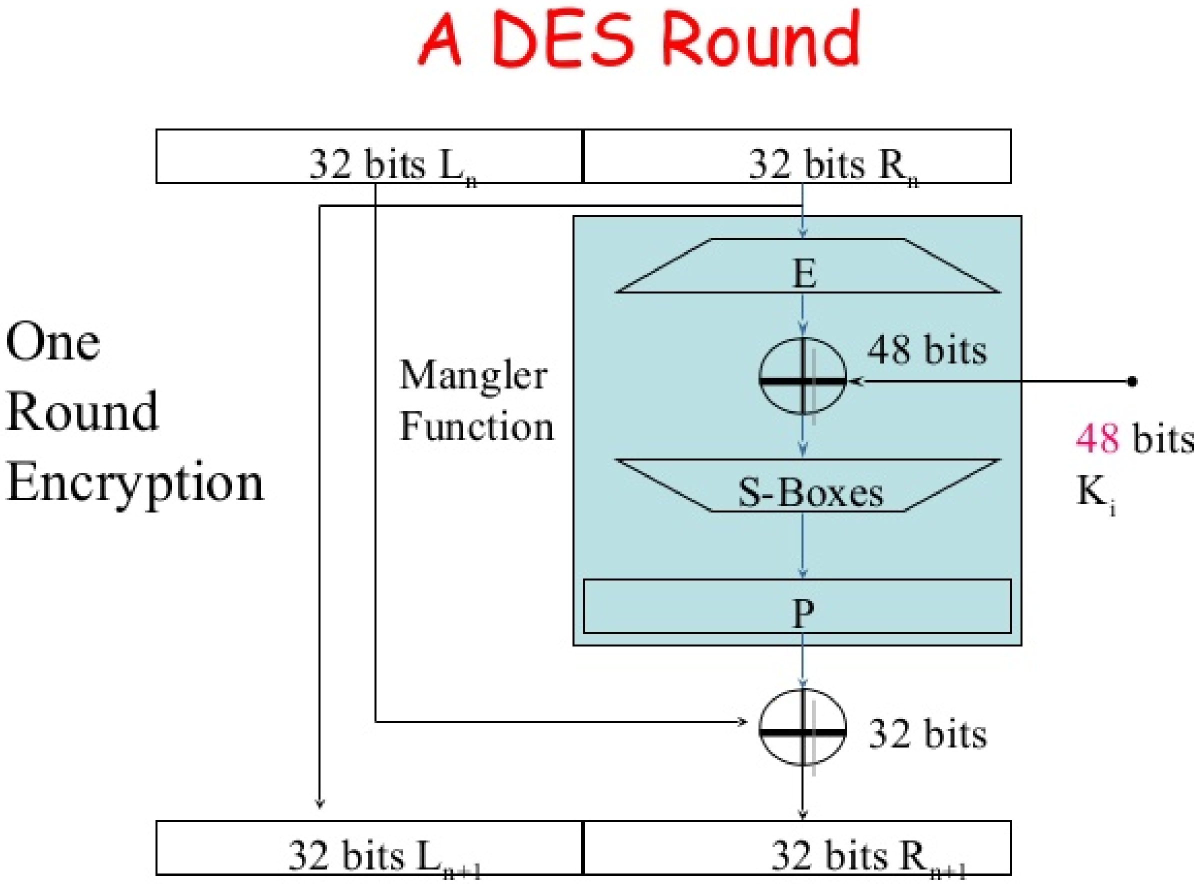
→R_{n+3} and R'_{n+3} differ on four bits since we have different inputs to two of the S-boxes

→$L_{n+3} = R'_{n+2}$ and $L_{n+2}' = R'_{n+2}$ now differ on two bits

- Seven rounds we expect all 32 bits in right half to be "affected" by input change

…

DES has sixteen rounds



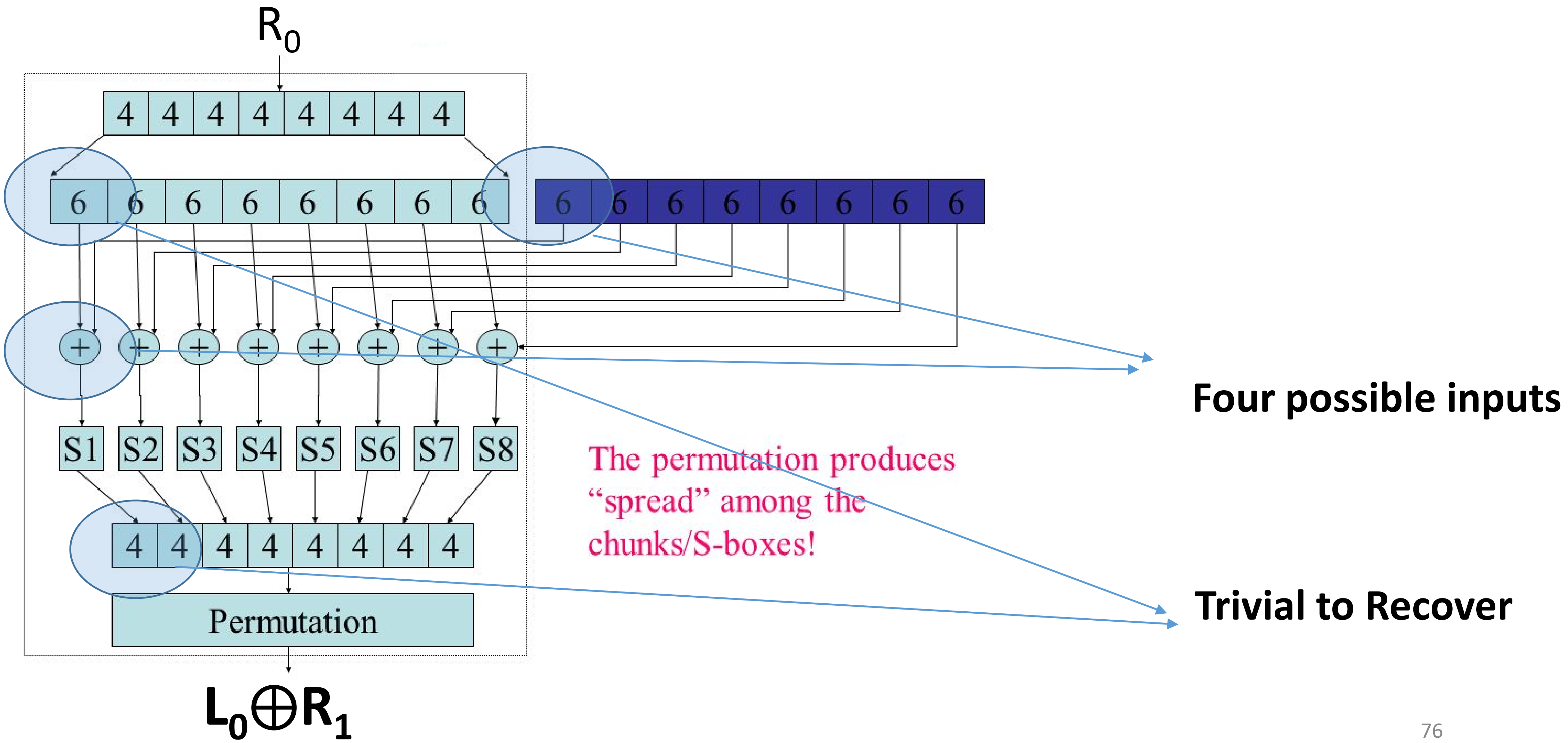
Attack on One-Round DES

- Given input output pair (x,y)
  - y=(L_1,R_1)
  - X=(L_0,R_0)
- Note: R_0=L_1
- Note: R_1=$L_0 \oplus f_1(R_0)$ where f_1 is the Mangling Function with key k_1

Conclusion:

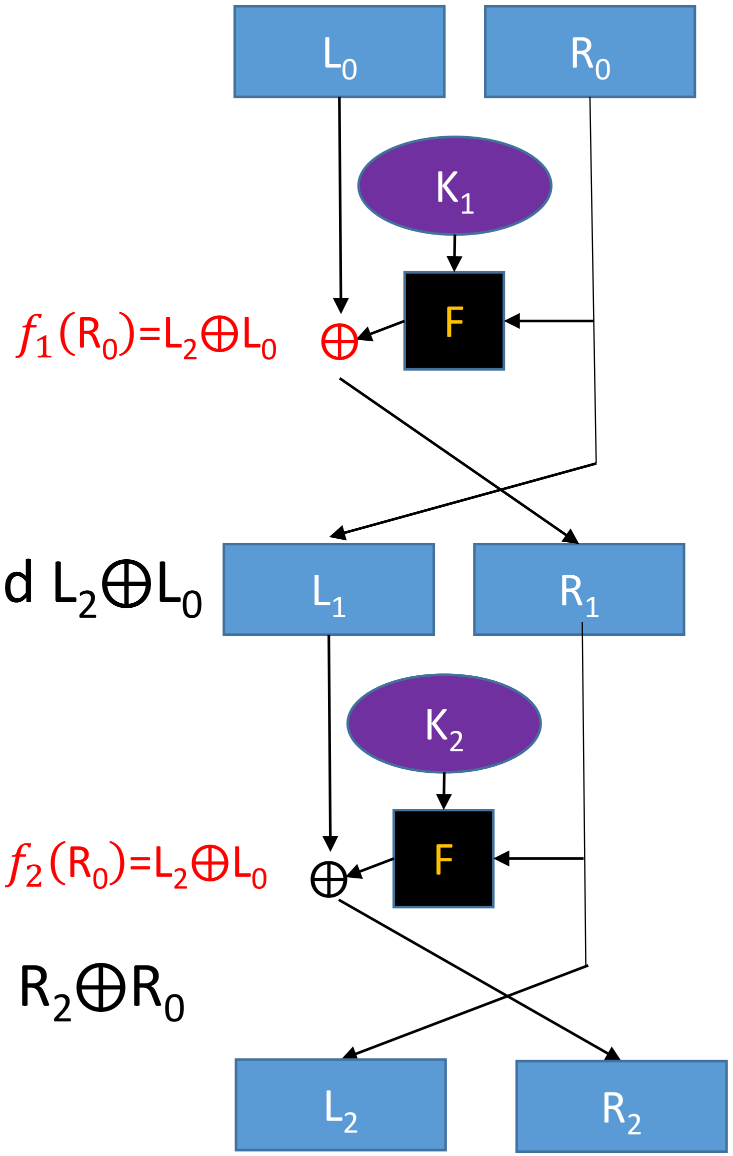
$$f_1(R_0)=L_0 \oplus R_1$$

Attack on One-Round DES



Attack on Two-Round DES

- Output y =(L_2,R_2)
- Note: R_1=$L_0 \oplus f_1(R_0)$
  - Also,R_1= L_2
  - Thus, $f_1(R_0)=L_2 \oplus L_0$
- So we can still attack the first round key k1 as before as R_0 and $L_2 \oplus L_0$ are known
- Note:$R_2=L_1 \oplus f_2(R_1)$
  - Also,L_1=R_0 and R_1= L_2
  - Thus, $f_2(L_2)=R_2 \oplus R_0$
- So we can attack the second round key k2 as before as L_2 and $R_2 \oplus R_0$ are known



Attack on Three-Round DES

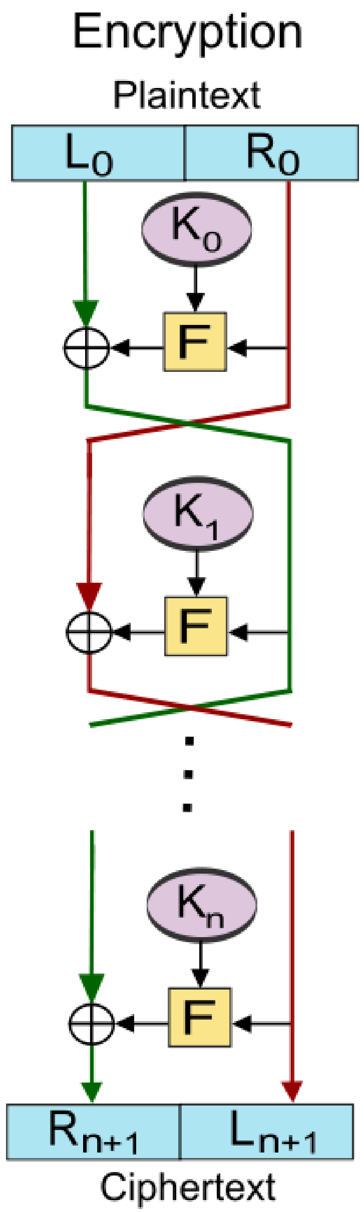
$$f_1(\mathbf{R_0}) \oplus f_3(\mathbf{R_2}) = (L_0 \oplus L_2) \oplus (L_2 \oplus R_3)$$
$$= L_0 \oplus R_3$$

We **know all of the values** L_0, R_0, R_3 and $L_3 = R_2$.

Leads to attack in time $\approx 2^{n/2}$

(See details in textbook)

Remember that DES is 16 rounds



DES Security

- Best Known attack is brute-force 2^{56}
  - Except under unrealistic conditions (e.g., 2^{43} known plaintexts)
- Brute force is not too difficult on modern hardware
- Attack can be accelerated further after precomputation
  - Output is a few terabytes
  - Subsequently keys are cracked in 2^{38} DES evaluations (minutes)
- Precomputation costs amortize over number of DES keys cracked
- Even in 1970 there were objections to the short key length for DES

Double DES

- Let $F_k(x)$ denote the DES block cipher

- A new block cipher F' with a key $k = (k_1, k_2)$ of length 2n can be defined by

$$F'_k(x) = F_{k_2}\left(F_{k_1}(x)\right)$$

- Can you think of an attack better than brute-force?

Meet in the Middle Attack

$$F'_k(x) = F_{k_2}\left(F_{k_1}(x)\right)$$

Goal: Given (x, $\mathrm{c} = F'_k(x)$) try to find secret key k in time and space $\mathrm{O}(n2^n)$.

- **Solution?**
  - **Key Observation**

$$F_{k_1}(x) = F_{k_2}^{-1}(\mathrm{c})$$

  - **Compute $F_K^{-1}(\mathrm{c})$ and $F_K(x)$ for each potential n-bit key K and store $\left(\boldsymbol{K}, F_K^{-1}(\mathrm{c})\right)$ and $\left(\boldsymbol{K}, F_K\ (\mathrm{x})\right)$**
  - **Sort each list of pairs (by $F_K^{-1}(\mathrm{c})$ or $F_K\ (\mathrm{x})$) to find K_1 and K_2.**

Triple DES Variant 1

- Let $F_k(x)$ denote the DES block cipher

- A new block cipher F' with a key $k = (k_1, k_2, k_3\ )$ of length 2n can be defined by

$$F'_k(x) = F_{k_3}\left(F_{k_2}^{-1}\left(F_{k_1}(x)\right)\right)$$

- Meet-in-the-Middle Attack Requires time $\Omega(2^{2n})$ and space $\Omega(2^{2n})$

Triple DES Variant 1

Allows backward compatibility with DES by setting $k_1=k_2=k_3$

- Let $F_k(x)$ denote the DES block cipher
- A new block cipher F' with a key $k = (k_1, k_2, k_3)$ of length 2n can be defined by

$$F'_k(x) = F_{k_3}\left(F_{k_2}^{-1}\left(F_{k_1}(x)\right)\right)$$

- Meet-in-the-Middle Attack Requires time $\Omega(2^{2n})$ and space $\Omega(2^{2n})$

Triple DES Variant 2

Just two keys!

- Let $F_k(x)$ denote the DES block cipher

- A new block cipher F' with a key $k = (k_1, k_2)$ of length 2n can be defined by

$$F'_k(x) = F_{k_1}\left(F_{k_2}^{-1}\left(F_{k_1}(x)\right)\right)$$

- Meet-in-the-Middle Attack still requires time $\Omega(2^{2n})$ and space $\Omega(2^{2n})$

- Key length is still just 112 bits (NIST recommends 128+ bits)

Triple DES Variant 1

$$F'_k(x) = F_{k_3}\left(F_{k_2}^{-1}\left(F_{k_1}(x)\right)\right)$$

- Standardized in 1999
- Still widely used, but it is relatively slow (three block cipher operations)
- Current gold standard: AES

Hash Functions from Block Ciphers

- Davies-Meyer Construction from block cipher F_K

$$H(K, x) = F_K(x)$$

Theorem: If $F: \{0,1\}^{\lambda} \times \{0,1\}^{\lambda} \rightarrow \{0,1\}^{\lambda}$ is modeled as an ideal block cipher then Davies-Meyer construction is a collision-resistant hash function

(**Concrete:** Need roughly $\mathrm{q} \approx 2^{\lambda/2}$ queries to find collision)

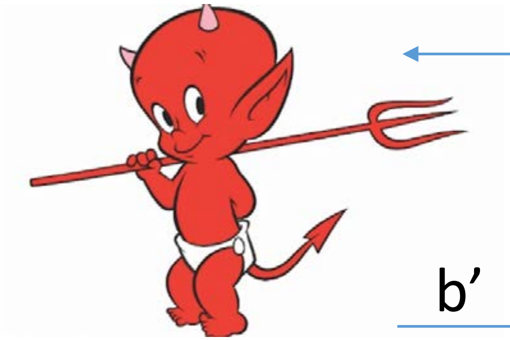
- **Ideal Cipher Model:** For each key K model F_K as a truly random permutation which may only be accessed in black box manner.
  - (Equivalent to Random Oracle Model)

Next Class

- Read Katz and Lindell 6.2.5-6.3
- AES & Differential Cryptanalysis

CS 555:Week 6: Topic 2
Stream Ciphers

PRG Security as a Game



R

b'

Random bit b

If b=1

$r \leftarrow \{0,1\}^n$

R = G(r)

Else

$0,1\}^{\ell(n)}$

ppt attacker

negligible function

$\forall$  $\Pr\Big[$  $Guesses\ b' = b\Big] \leq \frac{1}{2} + \mu(n)$

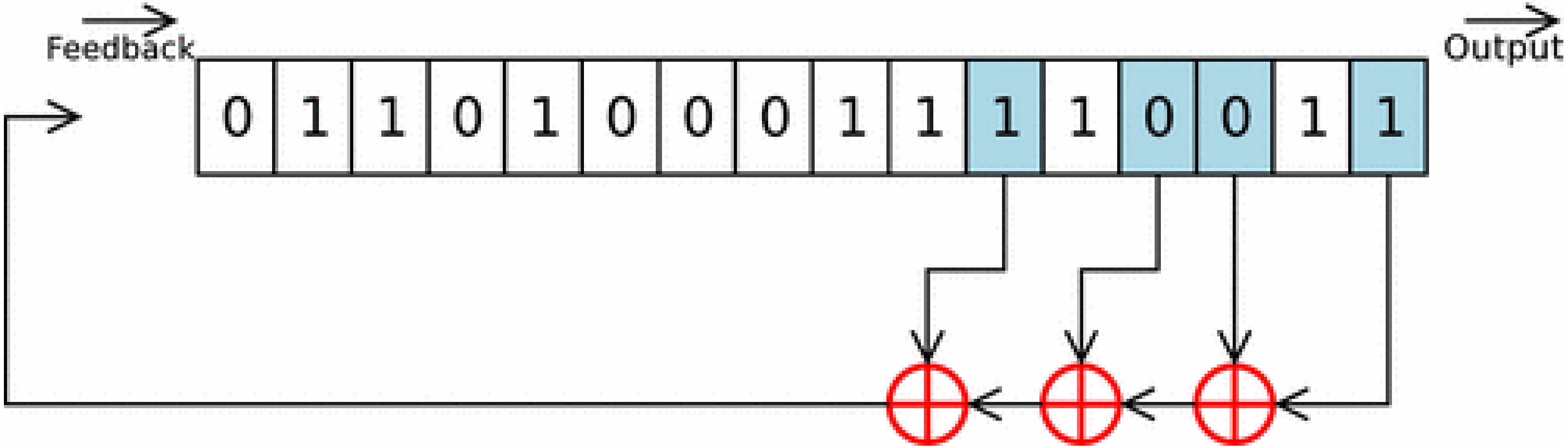
Stream Cipher vs PRG

- PRG pseudorandom bits output all at once

- Stream Cipher
  - Pseudorandom bits can be output as a stream
  - RC4, RC5 (Ron's Code)

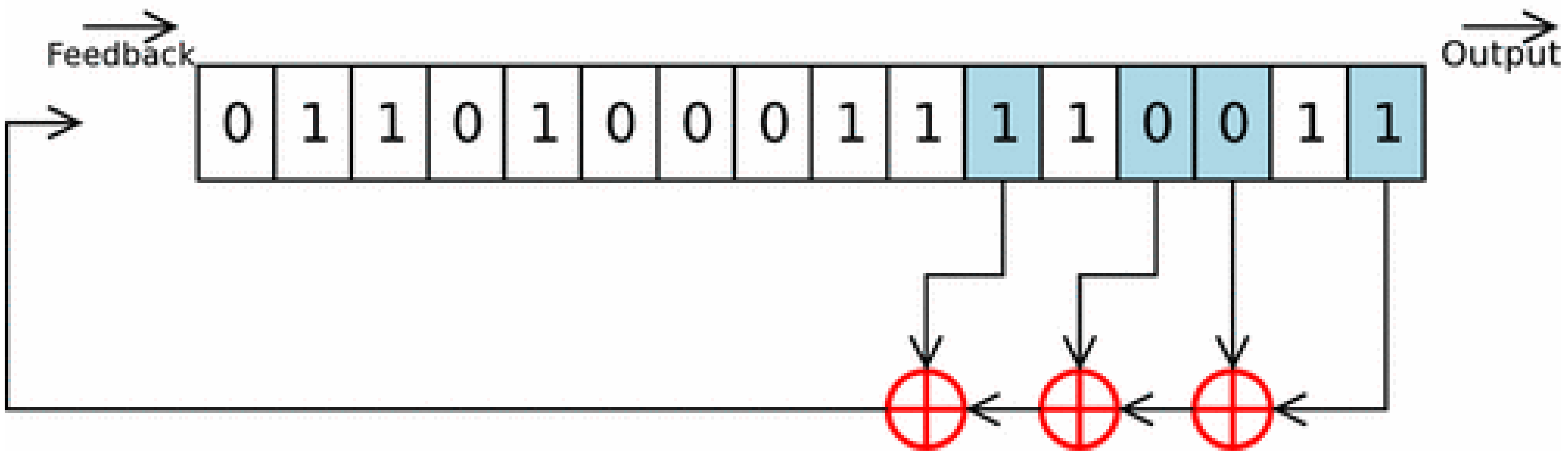
st_0 := Init(s)
For i=1 to ℓ:
    (y_i, st_i):=GetBits(st_{i-1})
Output: $y_1,\ldots,y_\ell$

Linear Feedback Shift Register



Linear Feedback Shift Register

- State at time t: $s_{n-1}^t, \dots, s_1^t, s_0^t$ (n registers)
- Feedback Coefficients: $\mathbf{S} \subseteq \{0, \dots, n\}$

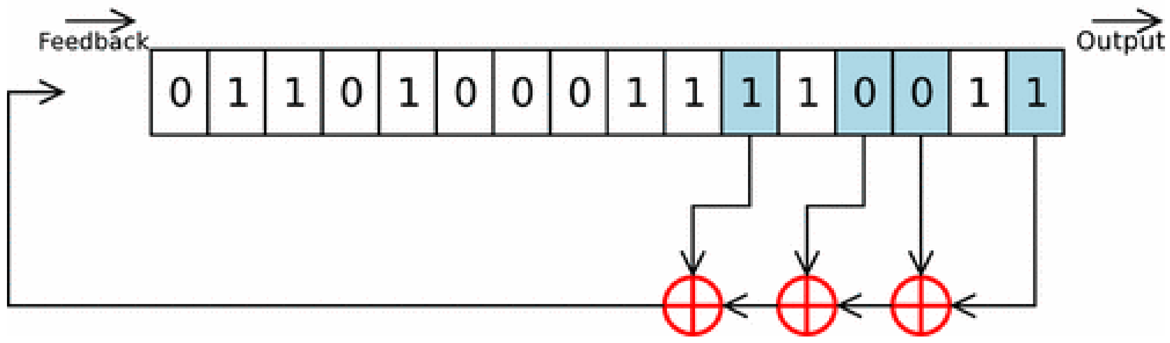


Linear Feedback Shift Register

- State at time t: $s_{n-1}^t, \dots, s_1^t, s_0^t$ (n registers)
- Feedback Coefficients: $\mathbf{S} \subseteq \{0, \dots, n-1\}$
- **State at time t+1:** $\bigoplus_{i \in S} s_i^t, s_{n-1}^t, \dots, s_1^t,$

$$s_{n-1}^{t+1} = \bigoplus_{i \in S} s_i^t, \qquad \text{and} \qquad s_i^{t+1} = s_{i+1}^t \text{ for i} < \text{n} - 1$$

Output at time t+1: $y_{t+1} = s_0^t$



Linear Feedback Shift Register

- **Observation 1**: First n bits of output reveal initial state

$$y_1, \dots, y_n = s_0^0, s_1^0, \dots, s_{n-1}^0$$

- **Observation 2**: Next n bits allow us to solve for n unknowns

$$x_i = \begin{cases} 1 & \text{if } i \in S \\ 0 & otherwise \end{cases}$$

$$y_{n+1} = y_n x_{n-1} + \cdots + y_1 x_0$$

Linear Feedback Shift Register

- **Observation 1**: First n bits of output reveal initial state

$$y_1, \dots, y_n = s_0^0, s_1^0, \dots, s_{n-1}^0$$

- **Observation 2**: Next n bits allow us to solve for n unknowns

$$x_i = \begin{cases} 1 & \text{if } i \in S \\ 0 & otherwise \end{cases}$$

$$y_{n+1} = y_n x_{n-1} + \cdots + y_1 x_0 \text{ mod } 2$$

Linear Feedback Shift Register

- **Observation 2**: Next n bits allow us to solve for n unknowns

$$x_i = \begin{cases} 1 & \text{if } i \in S \\ 0 & otherwise \end{cases}$$

$$\left.\begin{aligned} y_{n+1} &= y_n x_{n-1} + \cdots + y_1 x_0 \bmod 2 \\ &\vdots \\ y_{2n} &= y_{2n-1} x_{n-1} + \cdots + y_n x_0 \bmod 2 \end{aligned}\right\}$$

N unknowns &

N linear independent constraints

Removing Linearity

- Attacks exploited linear relationship between state and output bits

- **Nonlinear Feedback:**

Non linear function

$$\cancel{s_{n-1}^{t+1} = \bigoplus_{i \in S} s_i^t ,}$$

$$s_{n-1}^{t+1} = g(s_0^t, s_1^t, \ldots, s_{n-1}^t)$$

Removing Linearity

- Attacks exploited linear relationship between state and output bits
- **Nonlinear Combination:**

Non linear function

$$\cancel{\boldsymbol{y_{t+1} = s_0^t}}$$

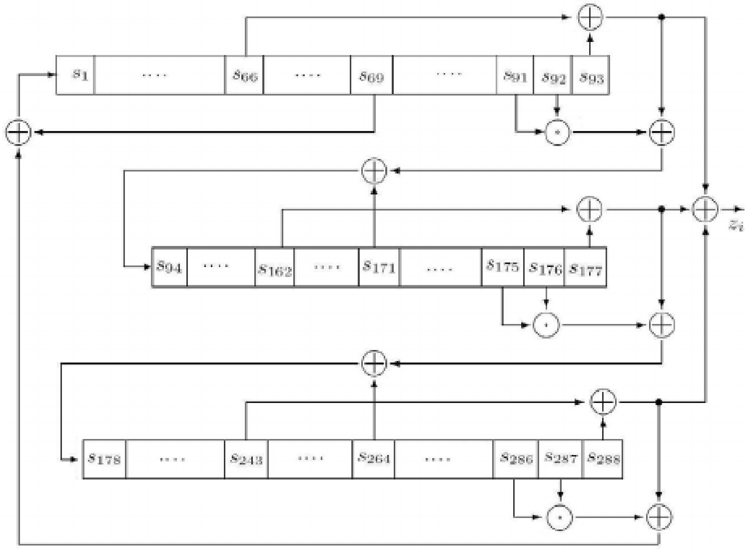
$$y_{t+1} = f(s_0^t, s_1^t, \dots, s_{n-1}^t)$$

- **Important**: f must be balanced!

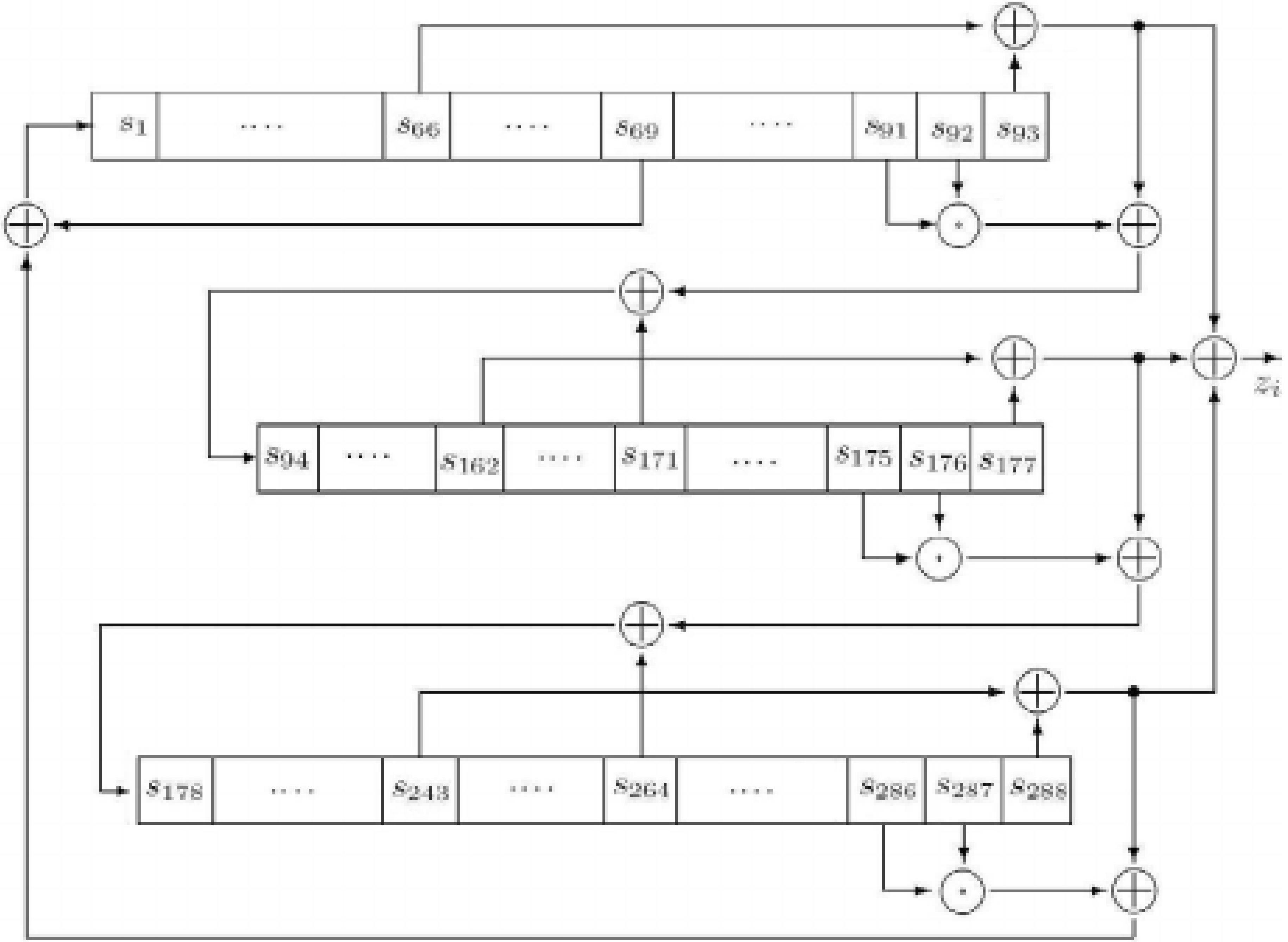
$$\Pr[f(x) = 1] \approx \frac{1}{2}$$

Trivium (2008)

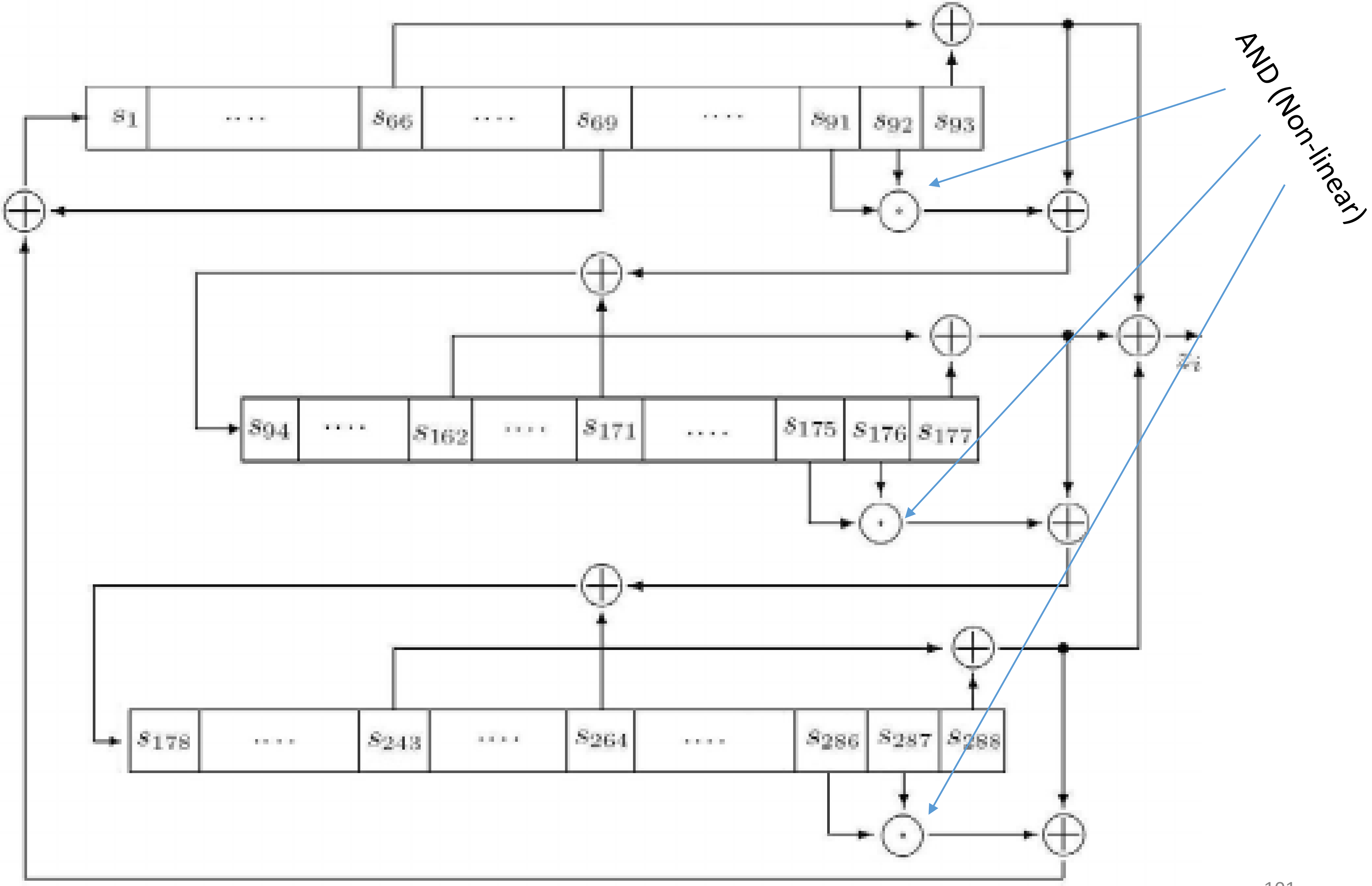
- Won the eSTREAM competition
- Currently, no known attacks are better than brute force
- Couples Output from three nonlinear Feedback Shift Registers
- First 4*288 “output bits” are discared



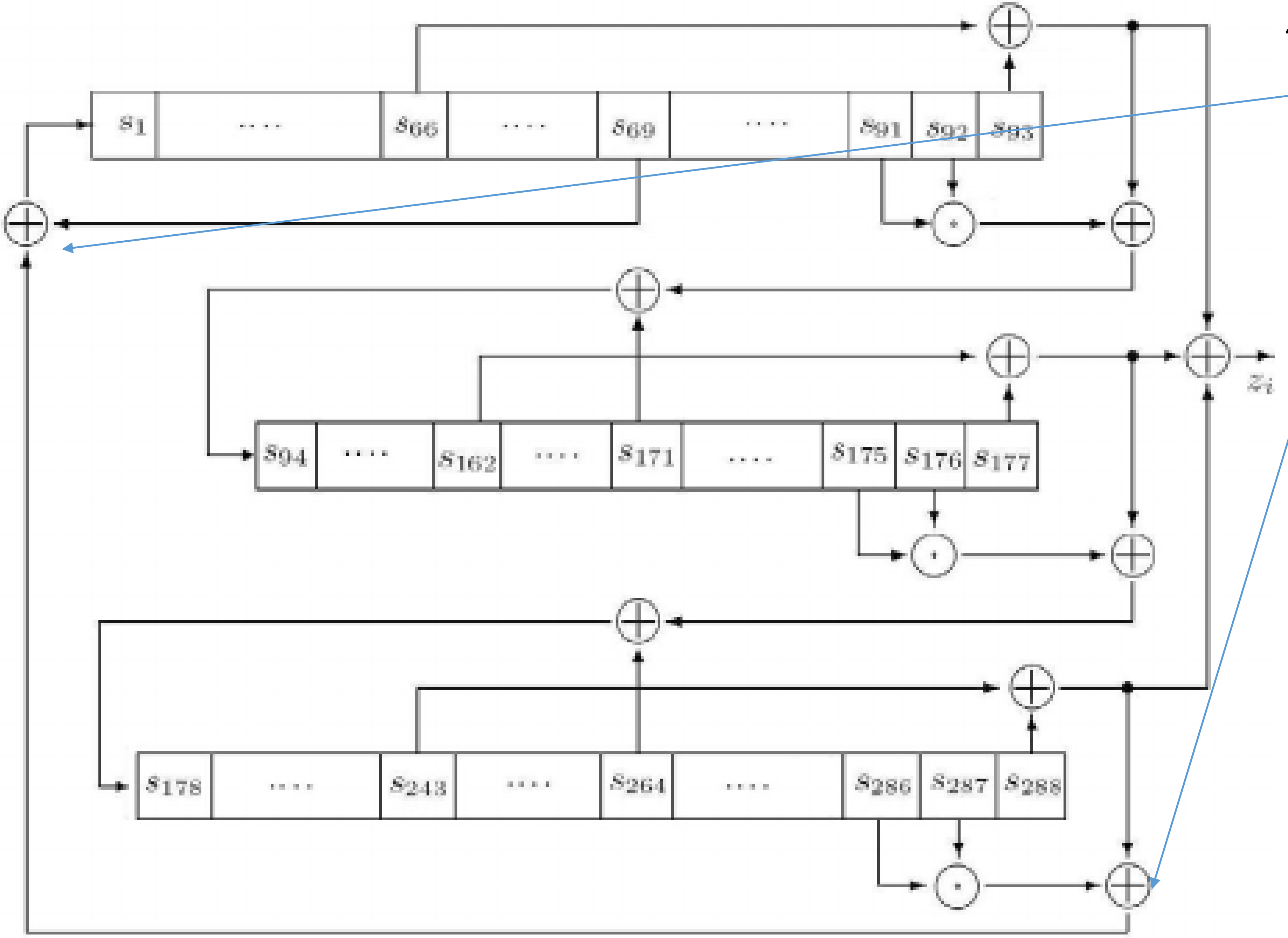
Trivium (2008)



Trivium (2008)



Trivium (2008)



Combination Generator

- Attacks exploited linear relationship between state and output bits

- **Nonlinear Combination:**

Non linear function

$$\cancel{y_{t+1} = s_0^t}$$

$$y_{t+1} = f(s_0^t, s_1^t, \ldots, s_{n-1}^t)$$

- **Important**: f must be balanced!

$$\Pr[f(x) = 1] \approx \frac{1}{2}$$

Feedback Shift Registers

- Good performance in hardware

- Performance is less ideal for software

The RC4 Stream Cipher

- A proprietary cipher owned by RSA, designed by Ron Rivest in 1987.
- Became public in 1994.
- Simple and effective design.
- Variable key size (typical 40 to 256 bits),
- Output unbounded number of bytes.
- Widely used (web SSL/TLS, wireless WEP).
- Extensively studied, not a completely secure PRNG, when used correctly, ~~no known attacks exist~~
- **Newer Versions**: RC5 and RC6
- **Rijndael** selected by NIST as AES in 2000

The RC4 Cipher

- The cipher internal state consists of
  - a 256-byte array S, which contains a permutation of 0 to 255
    - total number of possible states is $256! \approx 2^{1700}$
  - two indexes: i, j

```
i = j = 0
Loop
   i = (i + 1) (mod 256)
   j = (j + S[i]) (mod 256)
   swap(S[i], S[j])
   output S[S[i] + S[j] (mod 256)]
End Loop
```

Distinguishing Attack

- Let S_0 denote initial state
- Suppose that S_0[2]=0 and $S_0[1]= \mathrm{X} \neq 0$

	1	2	3	…	X	…	255
S_0	$S_0[1] \neq 0$	0	S_0[3]		S_0[X]		S_0[255]

```
i = j = 0
Loop
    i = (i + 1) (mod 256)
    j = (j + S[i]) (mod 256)
    swap(S[i], S[j])
    output S[S[i] + S[j] (mod 256)]
End Loop
```

Distinguishing Attack

- Let S_0 denote initial state
- Suppose that $S_0[2]$=0 and $S_0[1]= \mathrm{X} \neq 0$

	1	2	3	…	X	…	255
S_0	$\boldsymbol{X \neq 0}$	0	$S_0[3]$		$\boldsymbol{S_0}$**[X]**		$S_0[255]$

i=1, j =X

```
i = j = 0
Loop
   i = (i + 1) (mod 256)
   j = (j + S[i]) (mod 256)
   swap(S[i], S[j])
   output S[S[i] + S[j] (mod 256)]
End Loop
```

Distinguishing Attack

	1	2	3	…	X	…	255
S_0	$\boldsymbol{X \neq 0}$	0	$S_0[3]$		$\boldsymbol{S_0}$**[X]**		$S_0[255]$
S_1	$\boldsymbol{S_0}$**[X]**	0	$S_0[3]$		$\boldsymbol{X \neq 0}$		$S_0[255]$

i=1, j =X

Output y$_1$= S_1[S[i]+S[j]]

i=2, j =X

```
i = j = 0
Loop
    i = (i + 1) (mod 256)
    j = (j + S[i]) (mod 256)
    swap(S[i], S[j])
    output S[S[i] + S[j] (mod 256)]
End Loop
```

Distinguishing Attack

	1	2	3	…	X	…	255
S_0	$\boldsymbol{X \neq 0}$	0	$S_0[3]$		$\boldsymbol{S_0[X]}$		$S_0[255]$
S_1	$S_0[X]$	**0**	$S_0[3]$		$\boldsymbol{X \neq 0}$		$S_0[255]$
S_2	$S_0[X]$	$\boldsymbol{X \neq 0}$	$S_0[3]$		**0**		

i=2, j =X

```
i = j = 0
Loop
    i = (i + 1) (mod 256)
    j = (j + S[i]) (mod 256)
    swap(S[i], S[j])
    output S[S[i] + S[j] (mod 256)]
End Loop
```

Output:

$$y_2 = S_2[S_2[2]+S_2[X]]$$
$$= S_2[0+X]$$
$$=0$$

Distinguishing Attack

Let $\mathrm{p} = \Pr[S_0[2]$=0 and $S_0[1] \neq 2]$

$$p = \frac{1}{256}\left(1 - \frac{1}{255}\right)$$

- Probability second output byte is 0

$$\Pr[y_2 = 0 \mid S_0[2]\text{=0 and } S_0[1] \neq 2]p + \Pr[y_2 = 0 \mid S_0[2] \neq 0 \text{ or } S_0[1] \neq 2](1 - p)$$

$$= p + (1 - p)\frac{1}{256}$$

$$= \frac{1}{256}\left(1 - \frac{1}{255}\right) + \left(1 - \frac{1}{256} + \frac{1}{256}\frac{1}{255}\right)\frac{1}{256}$$

$$\approx \frac{2}{256}$$

Other Attacks

- Wired Equivalent Privacy (WEP) encryption used RC4 with an initialization vector

- Description of RC4 doesn't involve initialization vector…
  - But WEP imposes an initialization vector
  - K=IV || K'
  - Since IV is transmitted attacker may have first few bytes of K!

  - Giving the attacker partial knowledge of K often allows recovery of the entire key K' over time!