Block Ciphers

Recall that block ciphers are (practical implementations of) keyed pseudorandom permutations

They are **not** encryption schemes

Block Ciphers

Recall that block ciphers are (practical implementations of) keyed pseudorandom permutations

They are **not** encryption schemes

Nevertheless some terminology is also used for block ciphers:

- **Known plaintext attack**: The adversary knows x and $F_k(x)$, where x is not chosen by the attacker

Block Ciphers

Recall that block ciphers are (practical implementations of) keyed pseudorandom permutations

They are **not** encryption schemes

Nevertheless some terminology is also used for block ciphers:

- **Known plaintext attack**: The adversary knows x and $F_k(x)$, where x is not chosen by the attacker
- **Chosen plaintext attack**: The attacker can query F_k (with values of its choice)

Block Ciphers

Recall that block ciphers are (practical implementations of) keyed pseudorandom permutations

They are **not** encryption schemes

Nevertheless some terminology is also used for block ciphers:

- **Known plaintext attack**: The adversary knows x and $F_k(x)$, where x is not chosen by the attacker
- **Chosen plaintext attack**: The attacker can query F_k (with values of its choice)
- **Chosen ciphertext attack**: The attacker can query both F_k and F_k^{-1} (with values of its choice)

Designing Block Ciphers

- To design a block cipher, we want the computed function to be "indistinguishable" from a uniform permutation over $\{0,1\}^\ell$
- If x and x' differ, even just by one bit, the outputs of $F_k(x)$ and $F_k(x')$ should look unrelated (except for $F_k(x) \neq F_k(x')$)

Designing Block Ciphers

- To design a block cipher, we want the computed function to be "indistinguishable" from a uniform permutation over $\{0,1\}^\ell$

- If x and x' differ, even just by one bit, the outputs of $F_k(x)$ and $F_k(x')$ should look unrelated (except for $F_k(x) \neq F_k(x')$)

- On average $\approx \ell/2$ bits change between $F_k(x)$ and $F_k(x')$

- The position of the changing bits looks "random"

Designing Block Ciphers

- To design a block cipher, we want the computed function to be "indistinguishable" from a uniform permutation over $\{0,1\}^\ell$
- If x and x' differ, even just by one bit, the outputs of $F_k(x)$ and $F_k(x')$ should look unrelated (except for $F_k(x) \neq F_k(x')$)
- On average $\approx \ell/2$ bits change between $F_k(x)$ and $F_k(x')$
- The position of the changing bits looks "random"

How do we achieve this?

- Substitution Permutation Networks (SPNs)
- Feistel Networks

Designing Block Ciphers

- To design a block cipher, we want the computed function to be "indistinguishable" from a uniform permutation over $\{0,1\}^\ell$

- If x and x' differ, even just by one bit, the outputs of $F_k(x)$ and $F_k(x')$ should look unrelated (except for $F_k(x) \neq F_k(x')$)

- On average $\approx \ell/2$ bits change between $F_k(x)$ and $F_k(x')$

- The position of the changing bits looks "random"

How do we achieve this?

- Substitution Permutation Networks (SPNs)

- Feistel Networks

Substitution Permutation Networks (SPNs)

The input will be *mangled* in multiple steps

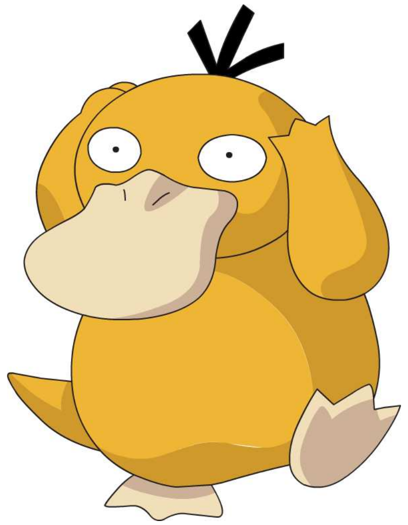
Two types of steps:

Substitution Permutation Networks (SPNs)

The input will be *mangled* in multiple steps

Two types of steps:

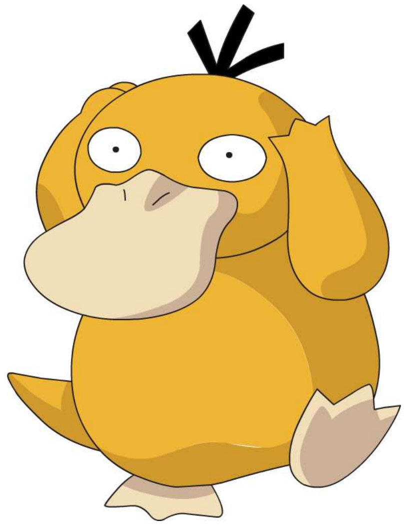
- **Confusion**: A small change in the input produces a small "random" change in the output



Substitution Permutation Networks (SPNs)

The input will be *mangled* in multiple steps

Two types of steps:



- **Confusion**: A small change in the input produces a small "random" change in the output

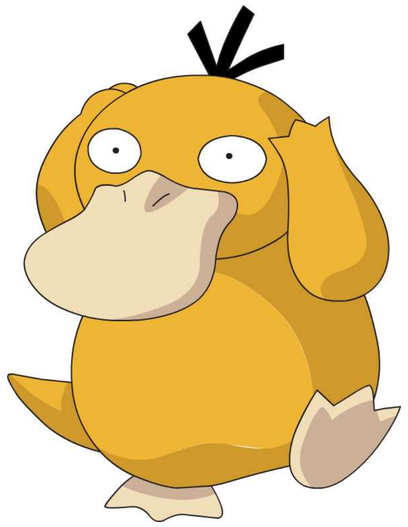
- **Diffusion:** The bits in the input are mixed so that a local change is spread throughout the block

Confusion

There are **many** random permutations

- Recall that $|\mathsf{Perm}_\ell| = (2^\ell)!$

- How many bits are needed to identify one of these permutations?

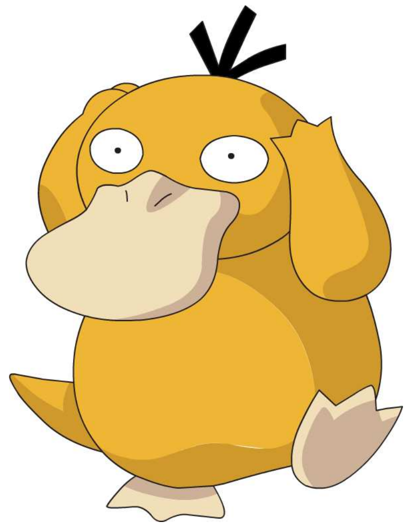


Confusion

There are **many** random permutations

- Recall that $|\mathsf{Perm}_\ell| = (2^\ell)!$
- How many bits are needed to identify one of these permutations?

$$\log(2^\ell!) \geq \log\left(\frac{2^\ell}{e}\right)^{2^\ell} = 2^\ell \cdot (\ell - \log_2 e)$$



Confusion

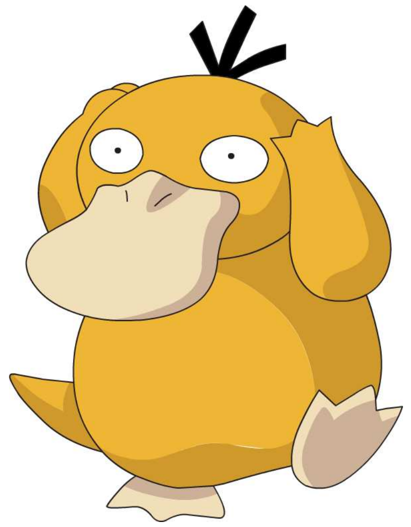
There are **many** random permutations

- Recall that $|\mathsf{Perm}_\ell| = (2^\ell)!$
- How many bits are needed to identify one of these permutations?

$$\log(2^\ell!) \geq \log\left(\frac{2^\ell}{e}\right)^{2^\ell} = 2^\ell \cdot (\ell - \log_2 e)$$

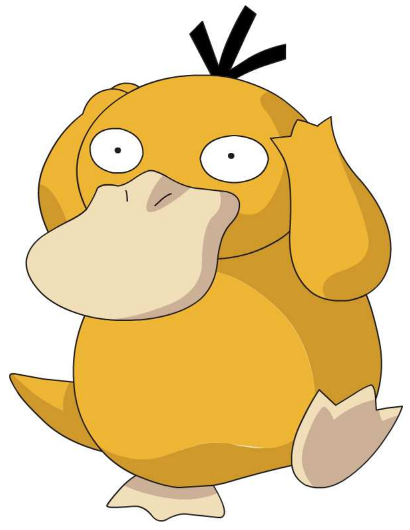
- Unfeasible even for small values of ℓ

Example: For block lengths of $\ell = 32$ bits, we would need ≈ 16GB



Confusion

There are **many** random permutations



- Recall that $|\mathsf{Perm}_\ell| = (2^\ell)!$
- How many bits are needed to identify one of these permutations?

$$\log(2^\ell!) \geq \log\left(\frac{2^\ell}{e}\right)^{2^\ell} = 2^\ell \cdot (\ell - \log_2 e)$$

- Unfeasible even for small values of ℓ

  Example: For block lengths of $\ell = 32$ bits, we would need ≈ 16GB

Idea: Build a "random" permutation on **long** inputs by using many "random" permutations on **short** inputs

Example: To store 8 permutations over $\{0,1\}^8$ we need less than $8 \cdot (8 \cdot 2^8)$ b $= 2$ KB

Confusion

Consider a keyed PRP F_k with a block length 64 bits defined as follows: (the length is just an example)

$$F_k(x) = f_{k_1}(x_1) \,\|\, f_{k_2}(x_2) \,\| f_{k_3}(x_3) \,\| \,\ldots\, \| \, f_{k_8}(x_8)$$

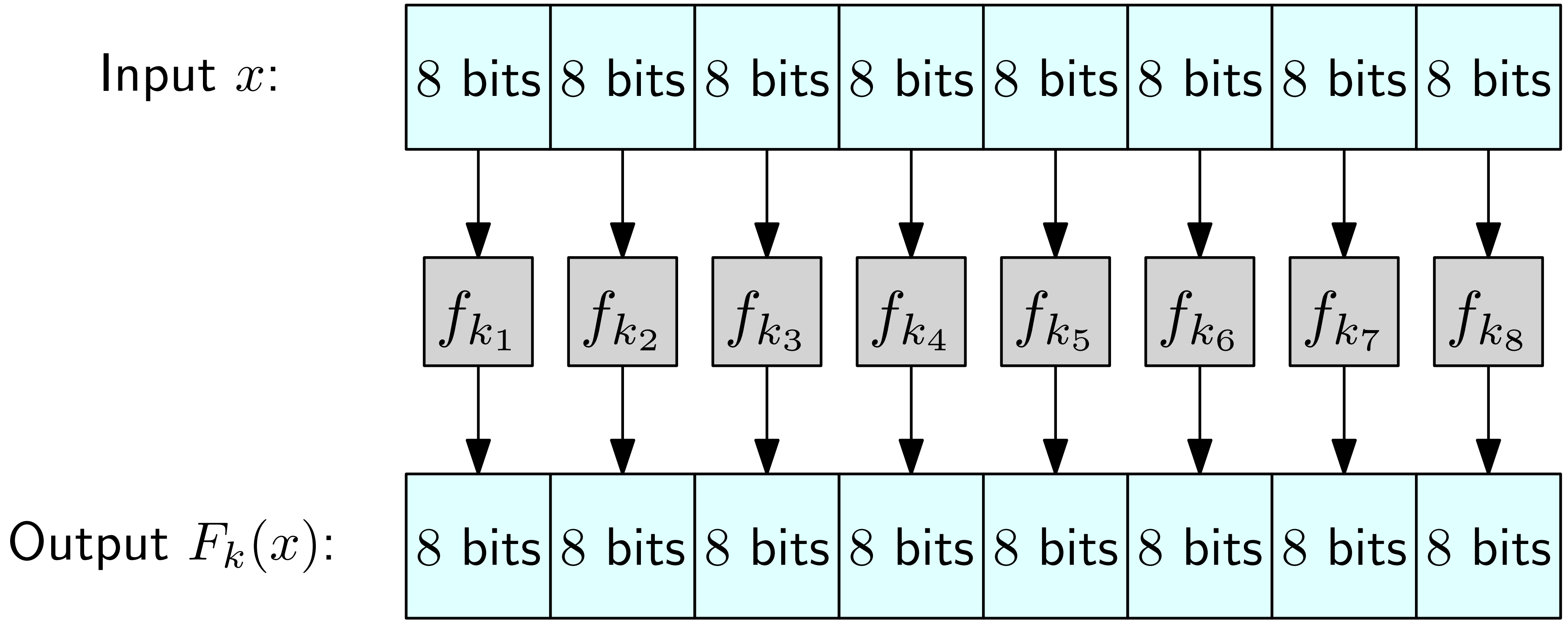
where $x = x_1 \| x_2 \| x_3 \| \ldots \| x_8$, $k = k_1 \| k_2 \| k_3 \| \ldots \| k_8$, all x_i are 8-bit long, and all f_{k_i} are permutations

Confusion

Consider a keyed PRP F_k with a block length 64 bits defined as follows: (the length is just an example)

$$F_k(x) = f_{k_1}(x_1) \parallel f_{k_2}(x_2) \parallel f_{k_3}(x_3) \parallel \dots \parallel f_{k_8}(x_8)$$

where $x = x_1 \| x_2 \| x_3 \| \dots \| x_8$, $k = k_1 \| k_2 \| k_3 \| \dots \| k_8$, all x_i are 8-bit long, and all f_{k_i} are permutations

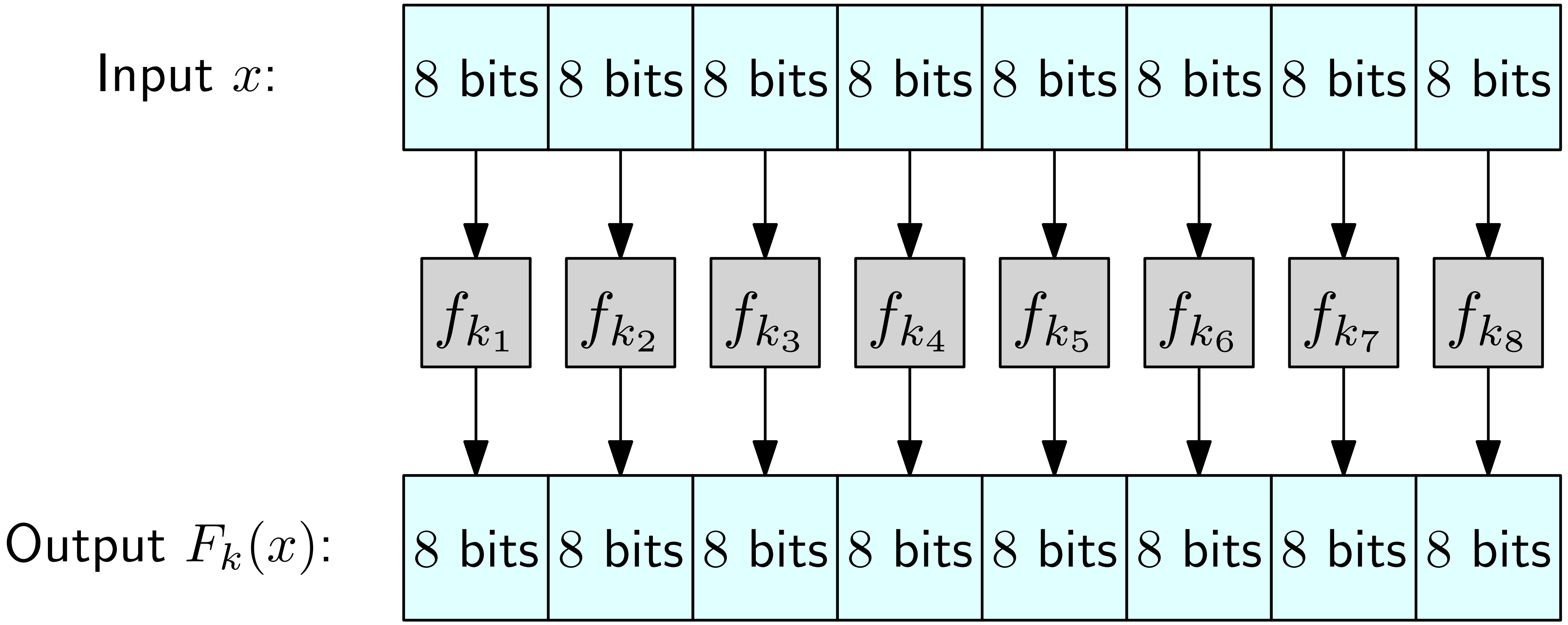


Confusion

Consider a keyed PRP F_k with a block length 64 bits defined as follows: (the length is just an example)

$$F_k(x) = f_{k_1}(x_1) \,\|\, f_{k_2}(x_2) \,\|\, f_{k_3}(x_3) \,\|\, \ldots \,\|\, f_{k_8}(x_8)$$

where $x = x_1\|x_2\|x_3\|\ldots\|x_8$, $k = k_1\|k_2\|k_3\|\ldots\|k_8$, all x_i are 8-bit long, and all f_{k_i} are permutations



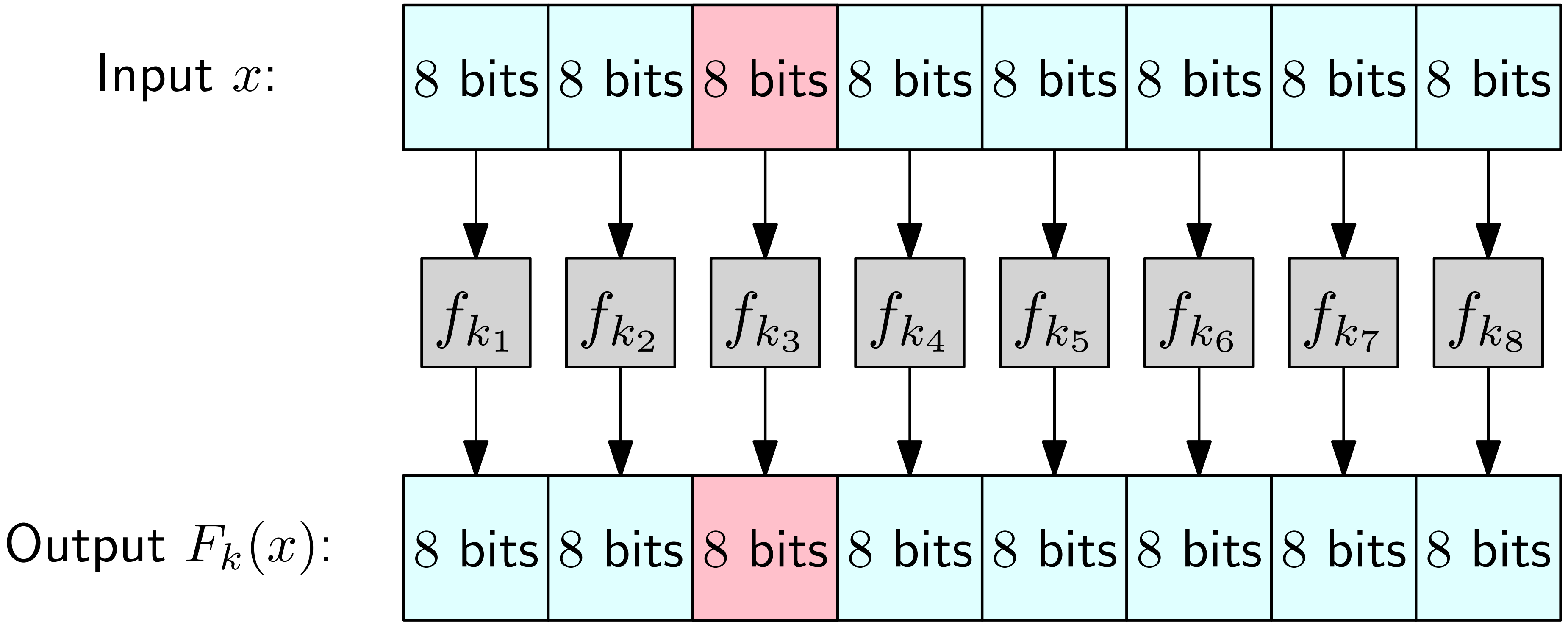
Is F a good PRP?

Confusion

Consider a keyed PRP F_k with a block length 64 bits defined as follows: (the length is just an example)

$$F_k(x) = f_{k_1}(x_1) \,\|\, f_{k_2}(x_2) \,\|\, f_{k_3}(x_3) \,\|\, \ldots \,\|\, f_{k_8}(x_8)$$

where $x = x_1\|x_2\|x_3\|\ldots\|x_8$, $k = k_1\|k_2\|k_3\|\ldots\|k_8$, all x_i are 8-bit long, and all f_{k_i} are permutations



Is F a good PRP?

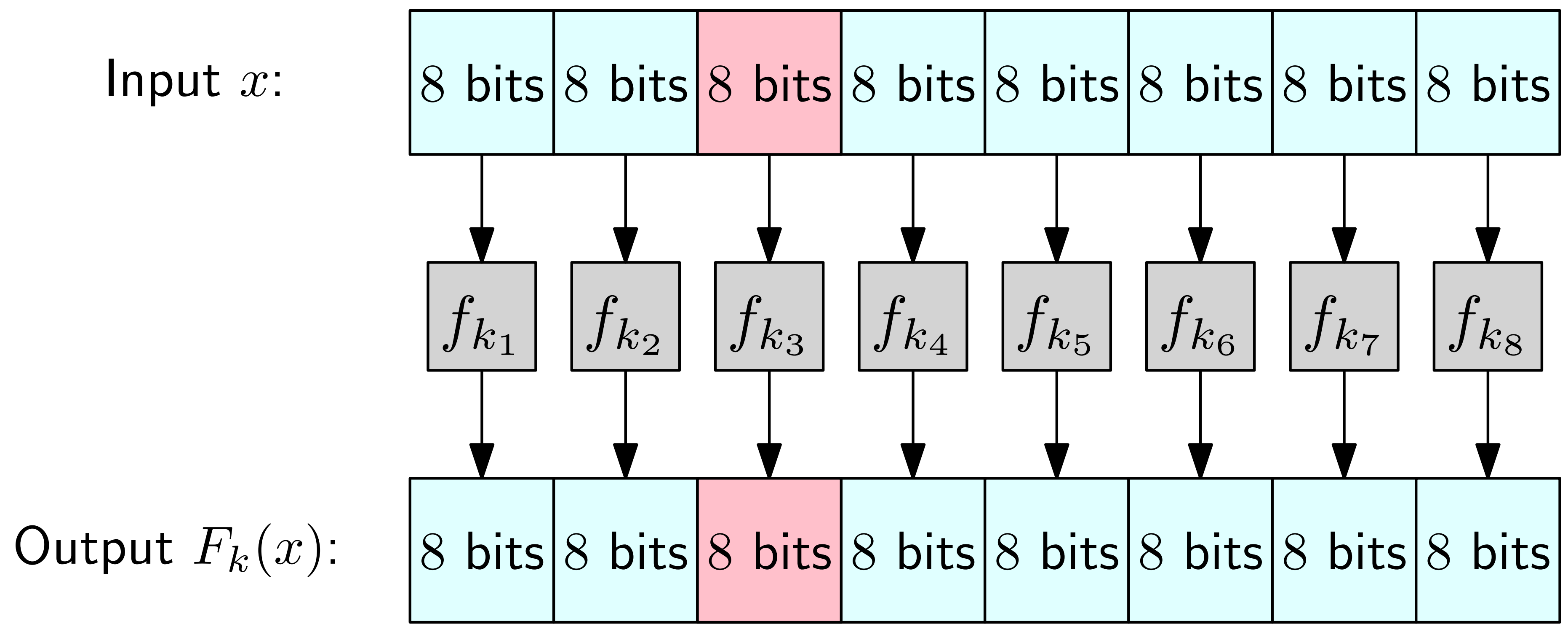
No! A local change in the input produces a local change in the output

Confusion

Consider a keyed PRP F_k with a block length 64 bits defined as follows: (the length is just an example)

$$F_k(x) = f_{k_1}(x_1) \,\|\, f_{k_2}(x_2) \,\|\, f_{k_3}(x_3) \,\|\, \ldots \,\|\, f_{k_8}(x_8)$$

where $x = x_1\|x_2\|x_3\|\ldots\|x_8$, $k = k_1\|k_2\|k_3\|\ldots\|k_8$, all x_i are 8-bit long, and all f_{k_i} are permutations



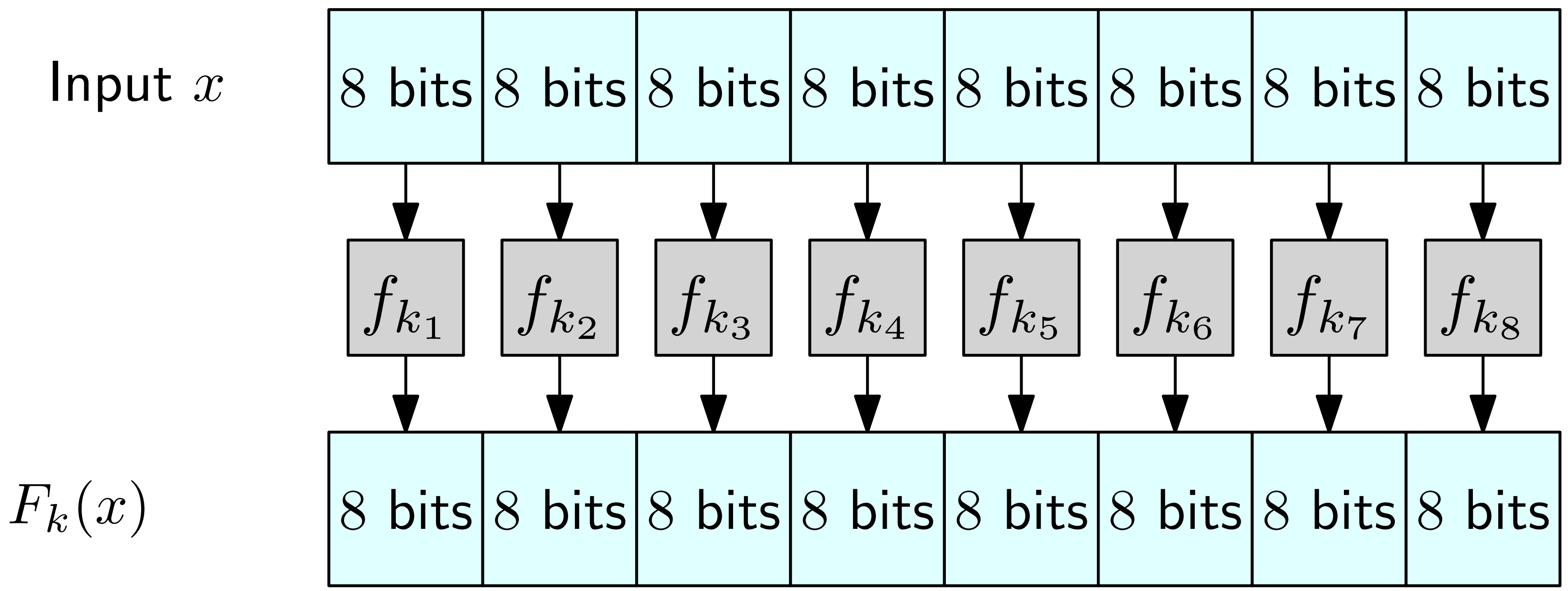
Is F a good PRP?

No! A local change in the input produces a local change in the output **Confusion but no diffusion**

Adding diffusion

We use a **mixing permutation** π to add diffusion

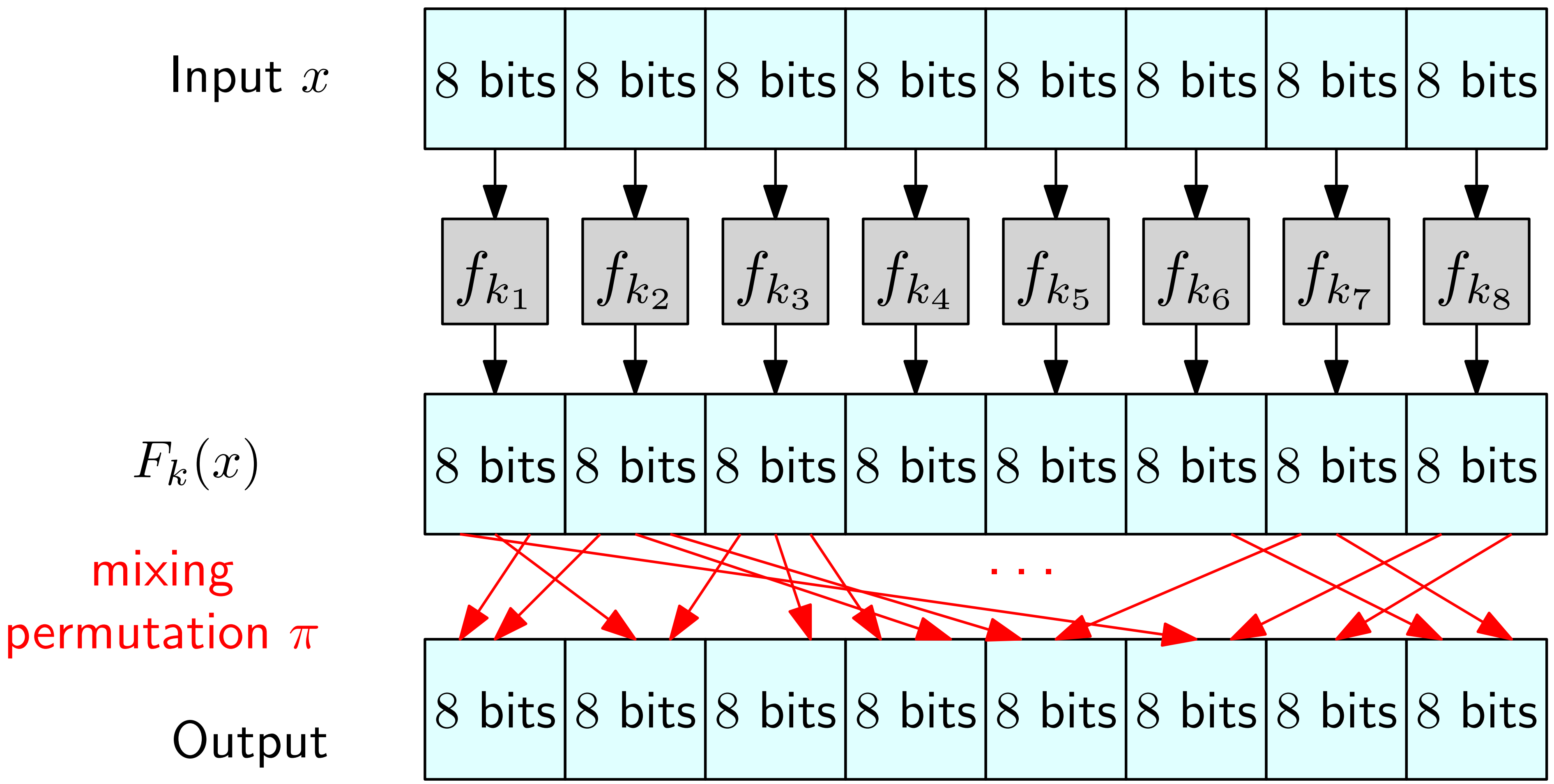
We move a generic bit in the i-th position of the input to the $\pi(i)$-th position of the output



Adding diffusion

We use a **mixing permutation** π to add diffusion

We move a generic bit in the i-th position of the input to the $\pi(i)$-th position of the output

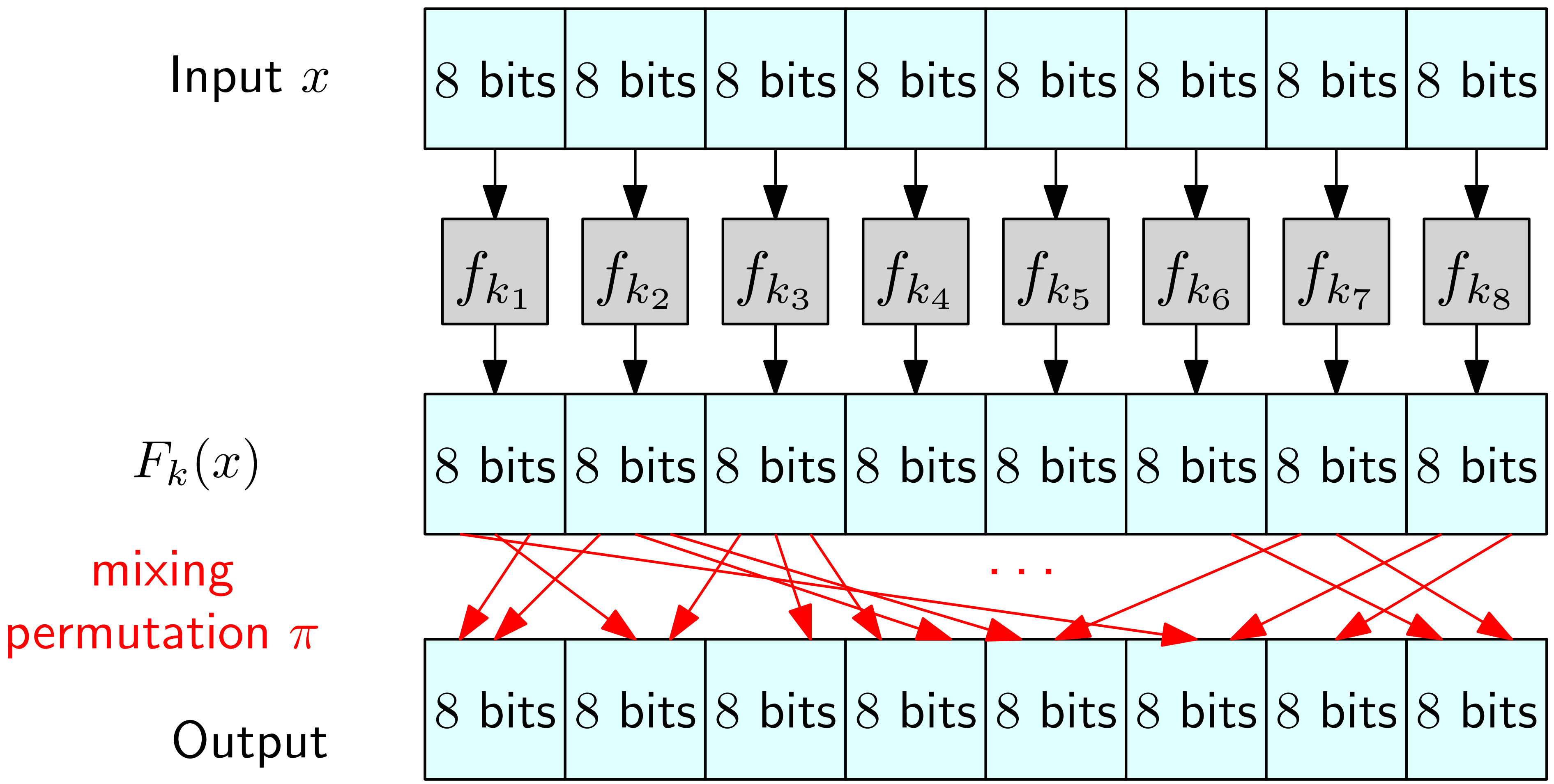


Adding diffusion

We use a **mixing permutation** π to add diffusion

We move a generic bit in the i-th position of the input to the $\pi(i)$-th position of the output

How many permutations π for block length ℓ? "Only" $\ell!$



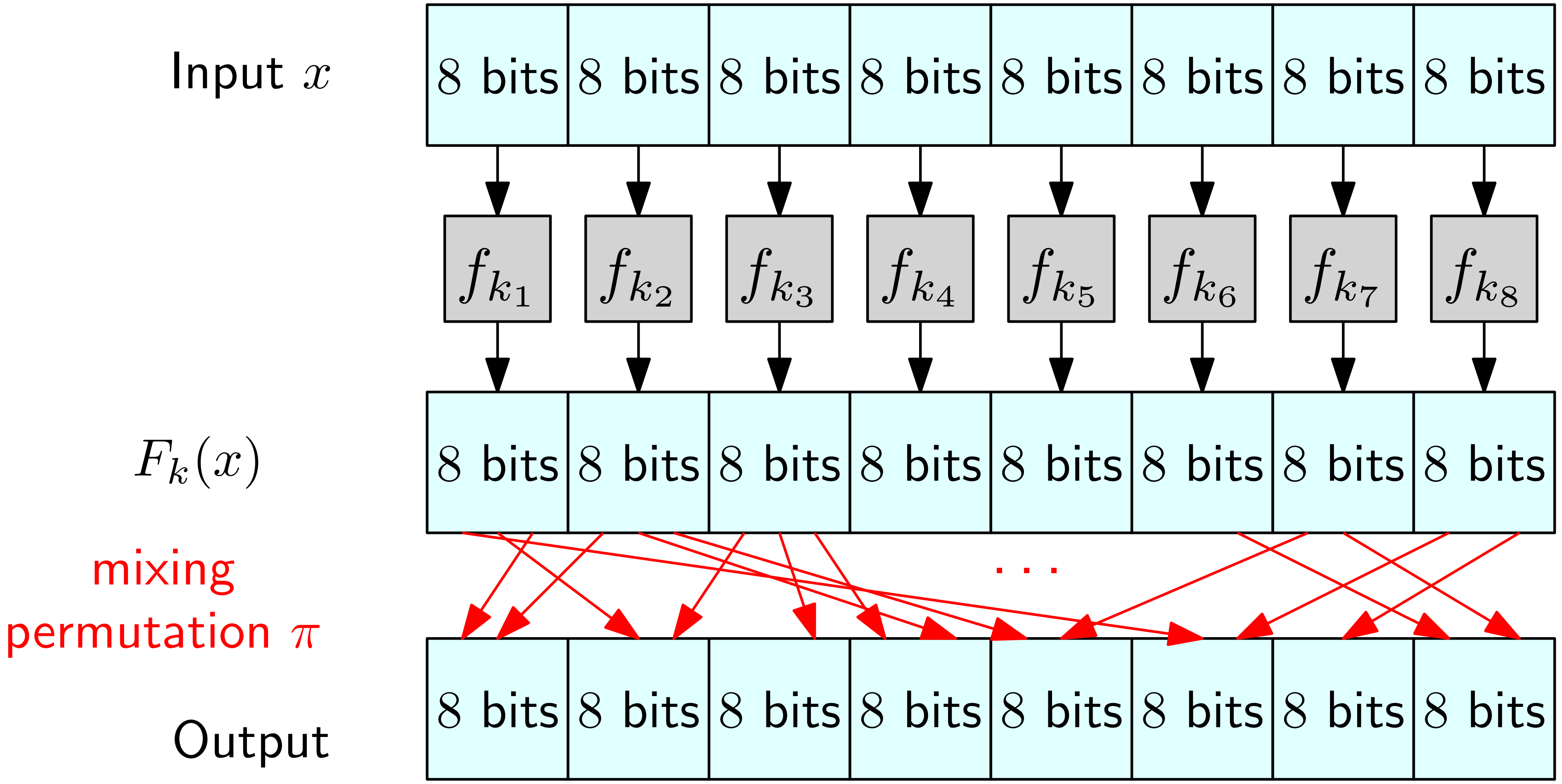
Adding diffusion

We use a **mixing permutation** π to add diffusion

We move a generic bit in the i-th position of the input to the $\pi(i)$-th position of the output

How many permutations π for block length ℓ? "Only" $\ell!$

Can be encoded using $\log \ell! \leq \ell \log \ell$ bits



Adding diffusion

We use a **mixing permutation** π to add diffusion

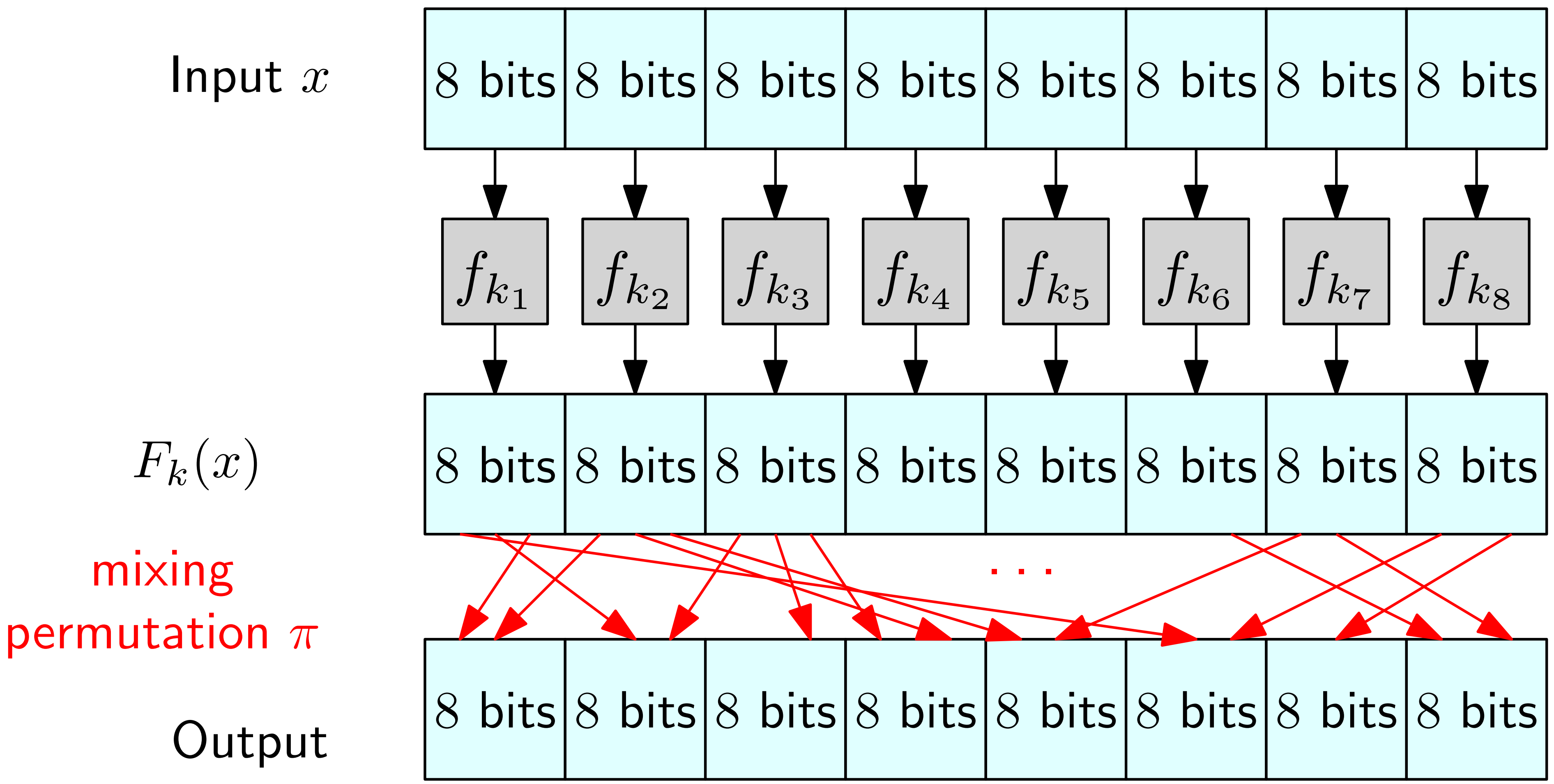
We move a generic bit in the i-th position of the input to the $\pi(i)$-th position of the output

How many permutations π for block length ℓ?

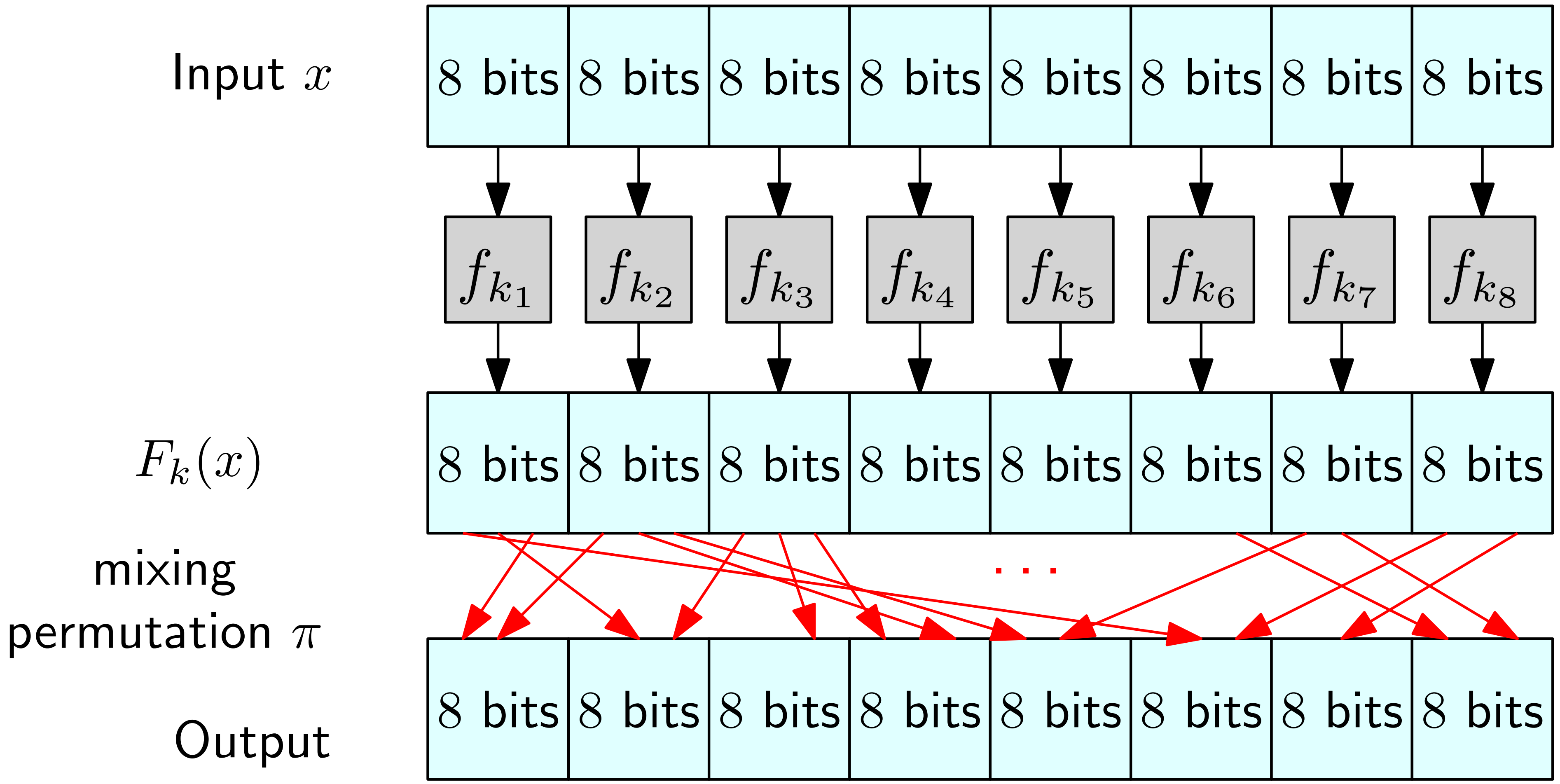
"Only" $\ell!$

Can be encoded using $\log \ell! \leq \ell \log \ell$ bits

In practice the mixing permutation does not depend on the key and is carefully designed and **fixed**

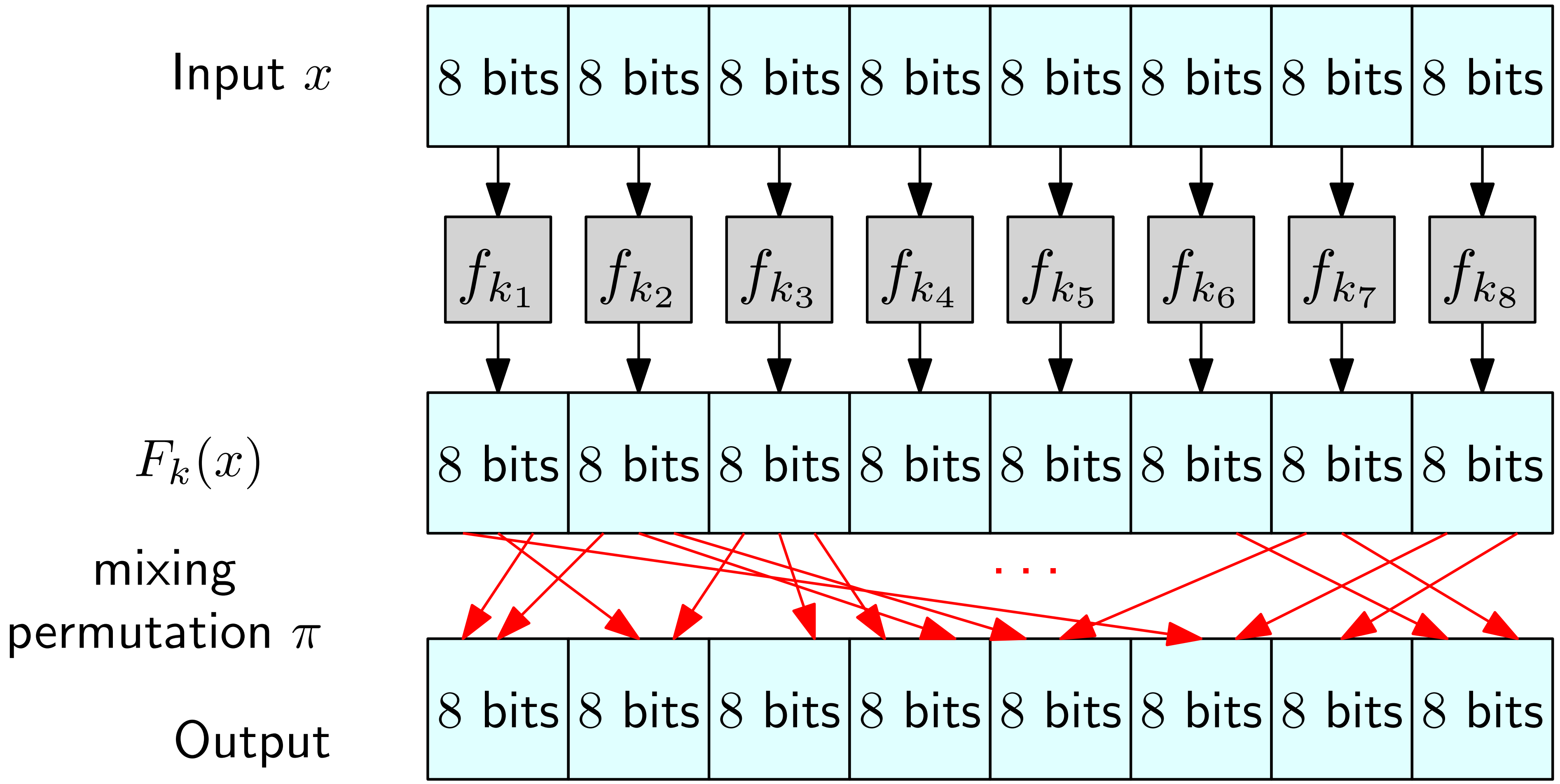


We have a Substitution Permutation Network (SPN)



Is this a PRP (i.e., is this invertible)?

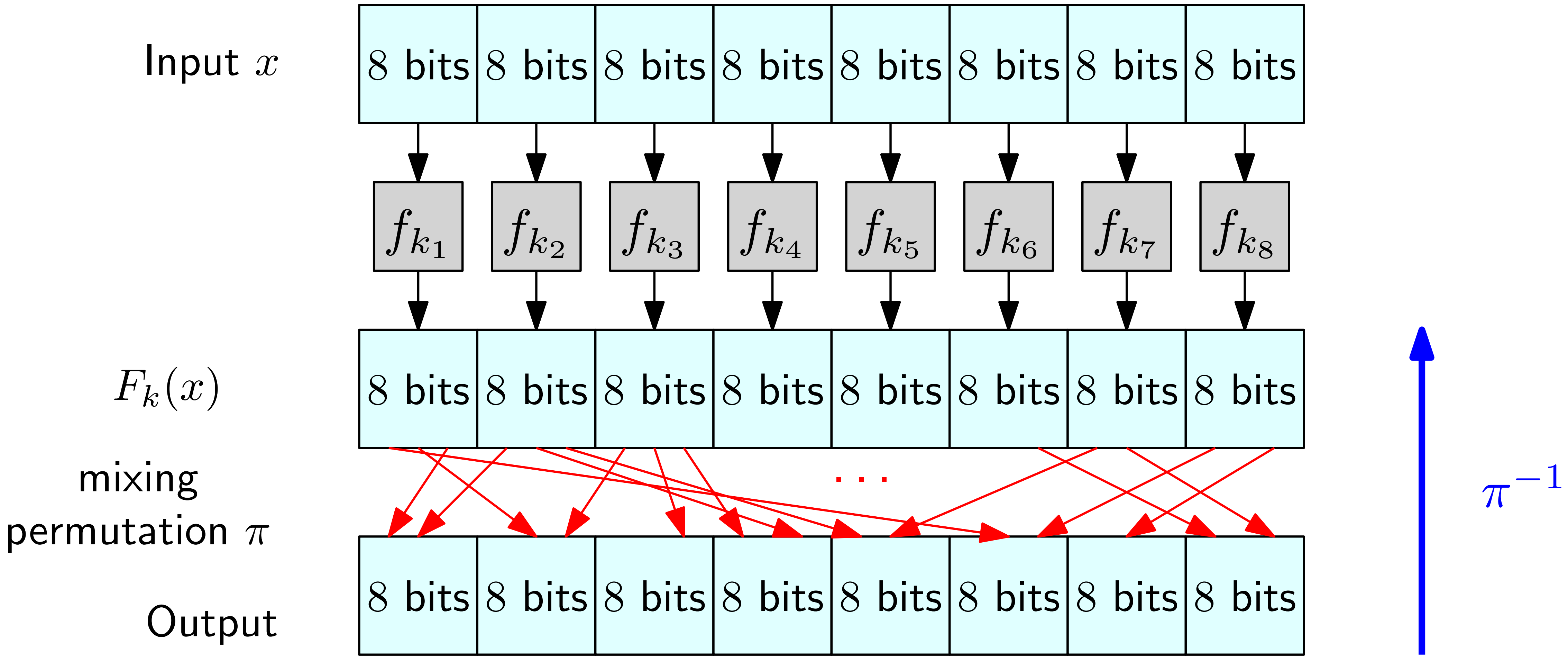
We have a Substitution Permutation Network (SPN)



Is this a PRP (i.e., is this invertible)?

Yes, proceed backwards:

We have a Substitution Permutation Network (SPN)

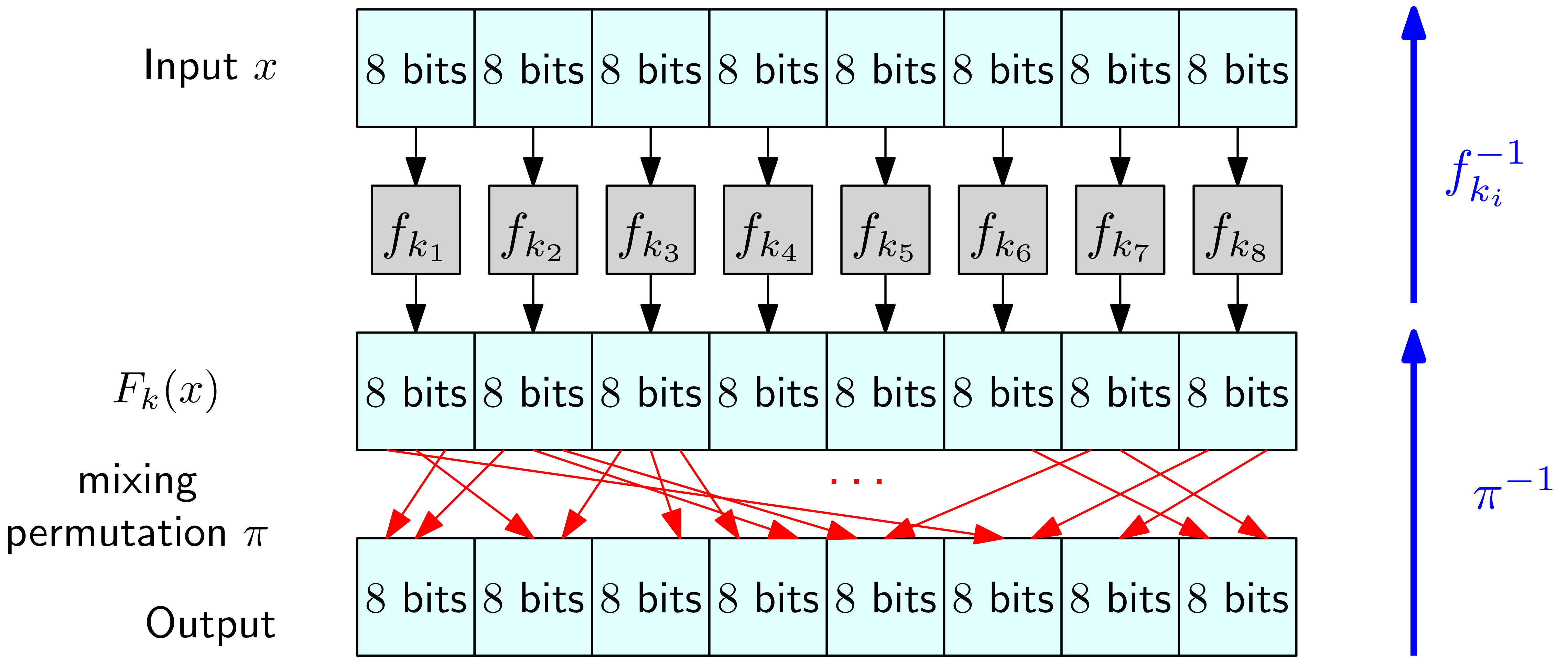


Is this a PRP (i.e., is this invertible)?

Yes, proceed backwards:

- The mixing permutation is... a permutation, and hence invertible

We have a Substitution Permutation Network (SPN)

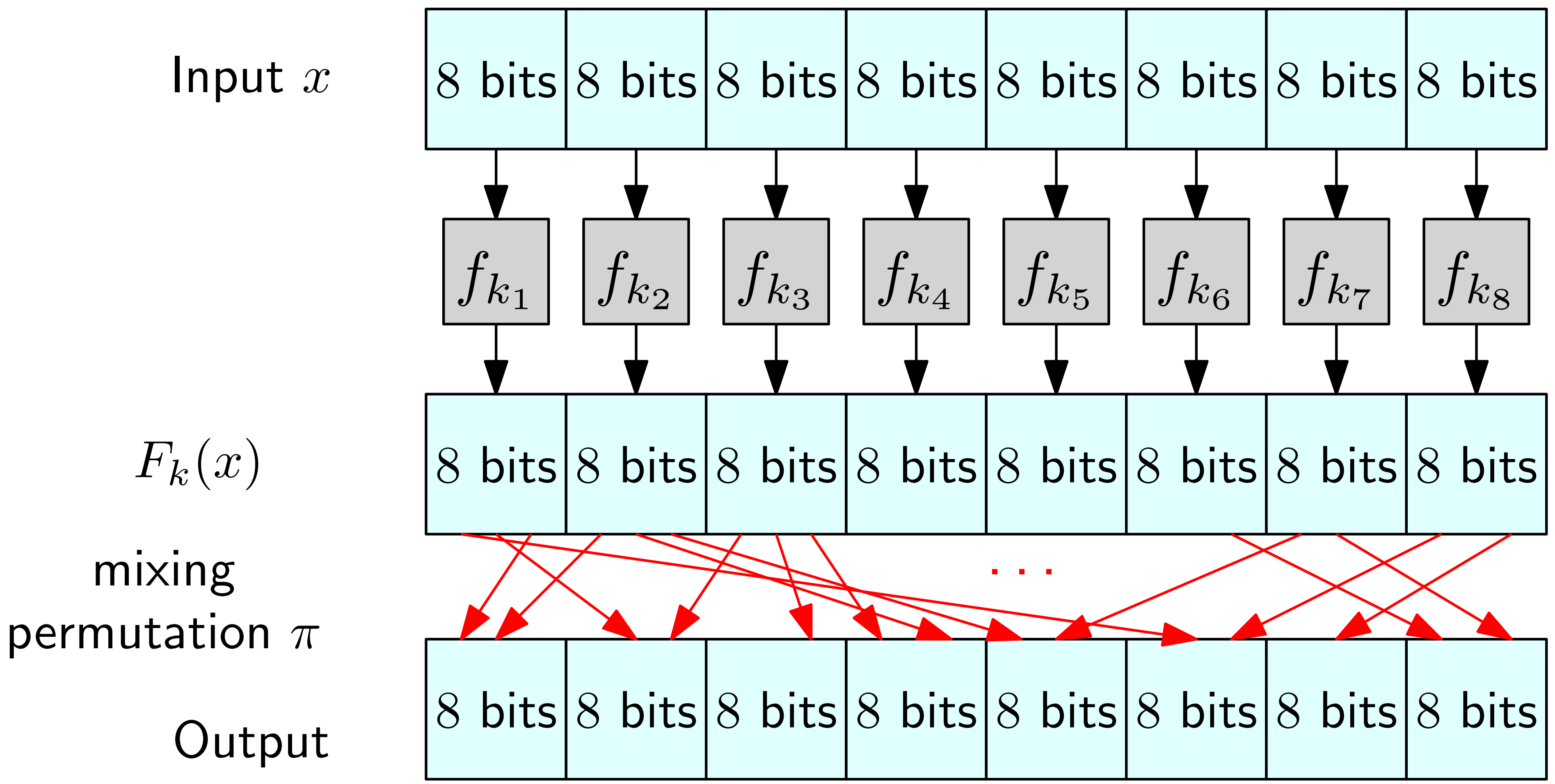


Is this a PRP (i.e., is this invertible)?

Yes, proceed backwards:

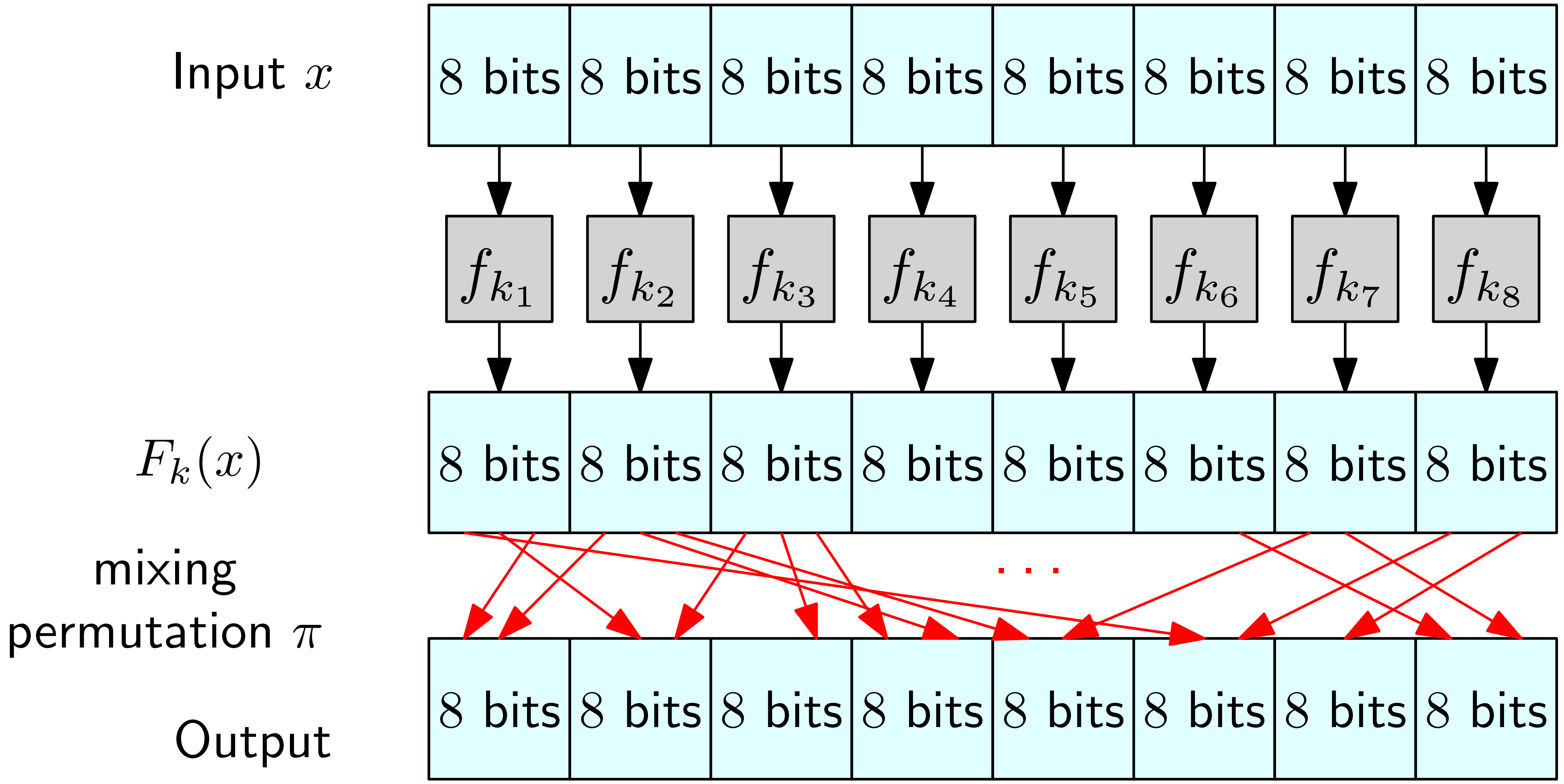
- The mixing permutation is... a permutation, and hence invertible
- Each function f_{k_i} is also a permutation, and hence invertible

Substitution Permutation Networks



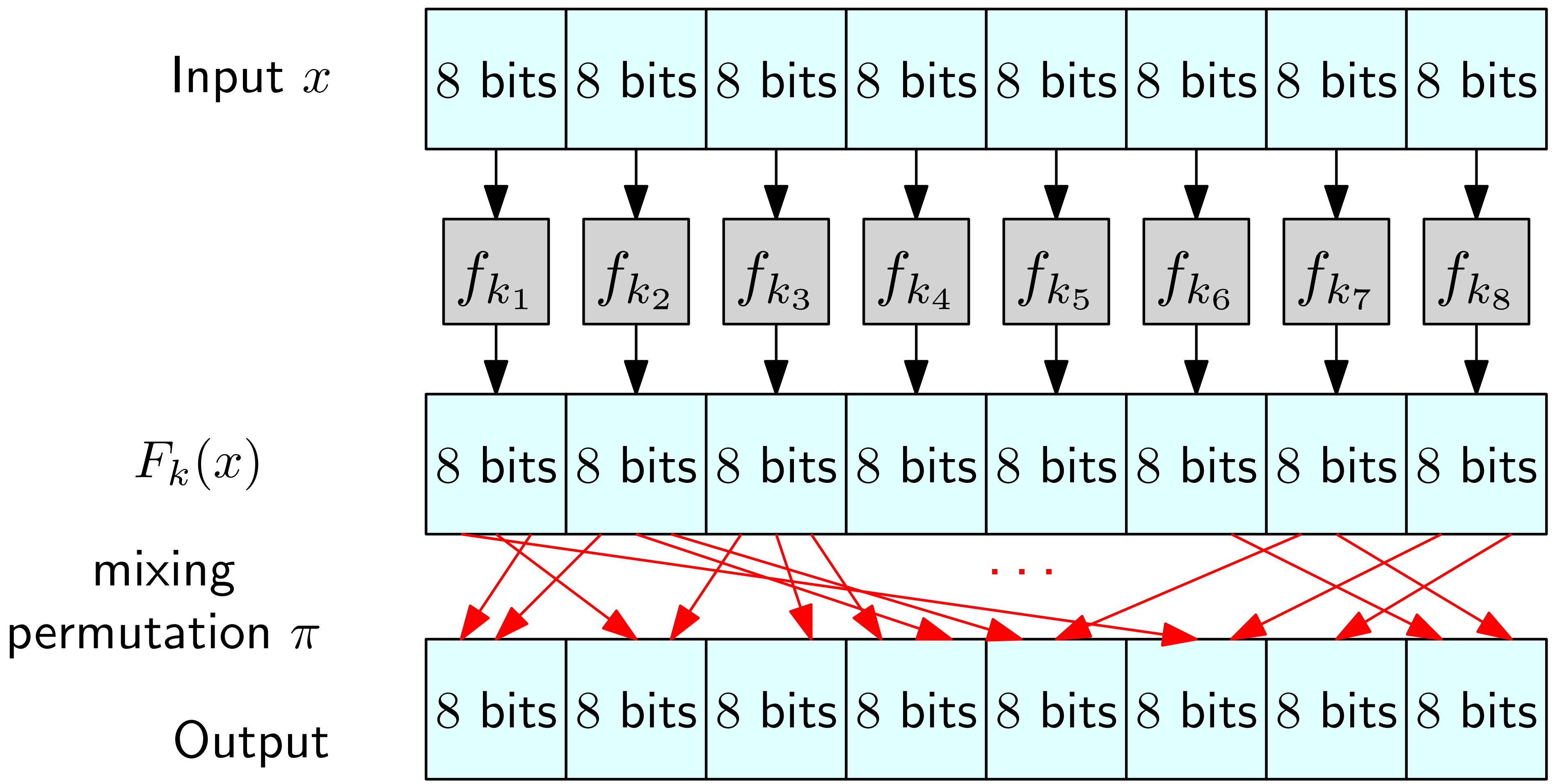
Is the function computed by this SPN a **good** PRP?

Substitution Permutation Networks



Is the function computed by this SPN a **good** PRP? **No**

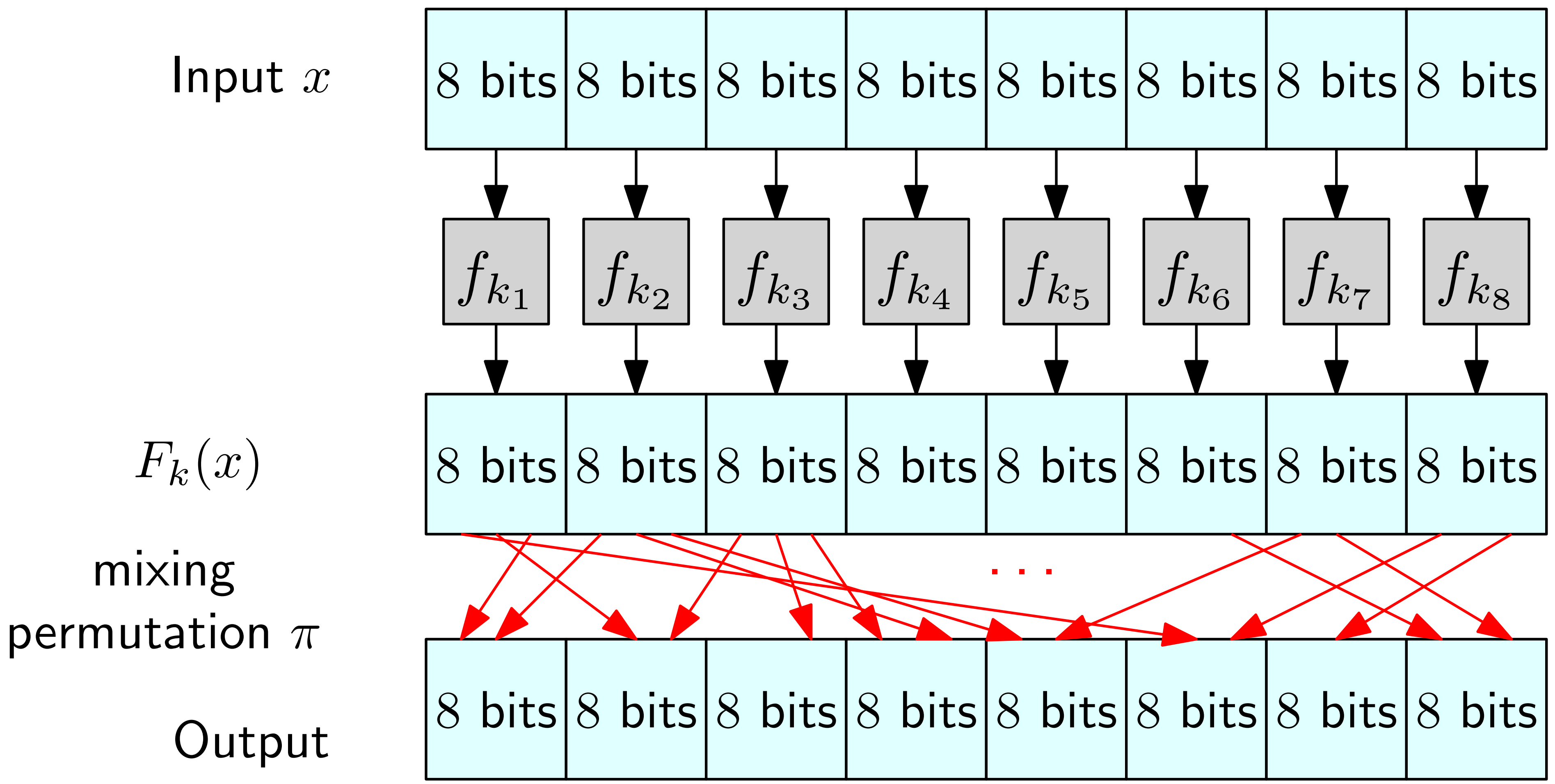
Substitution Permutation Networks



Is the function computed by this SPN a **good** PRP? **No**

- The mixing permutation is fixed. An adversary can always undo the last step!

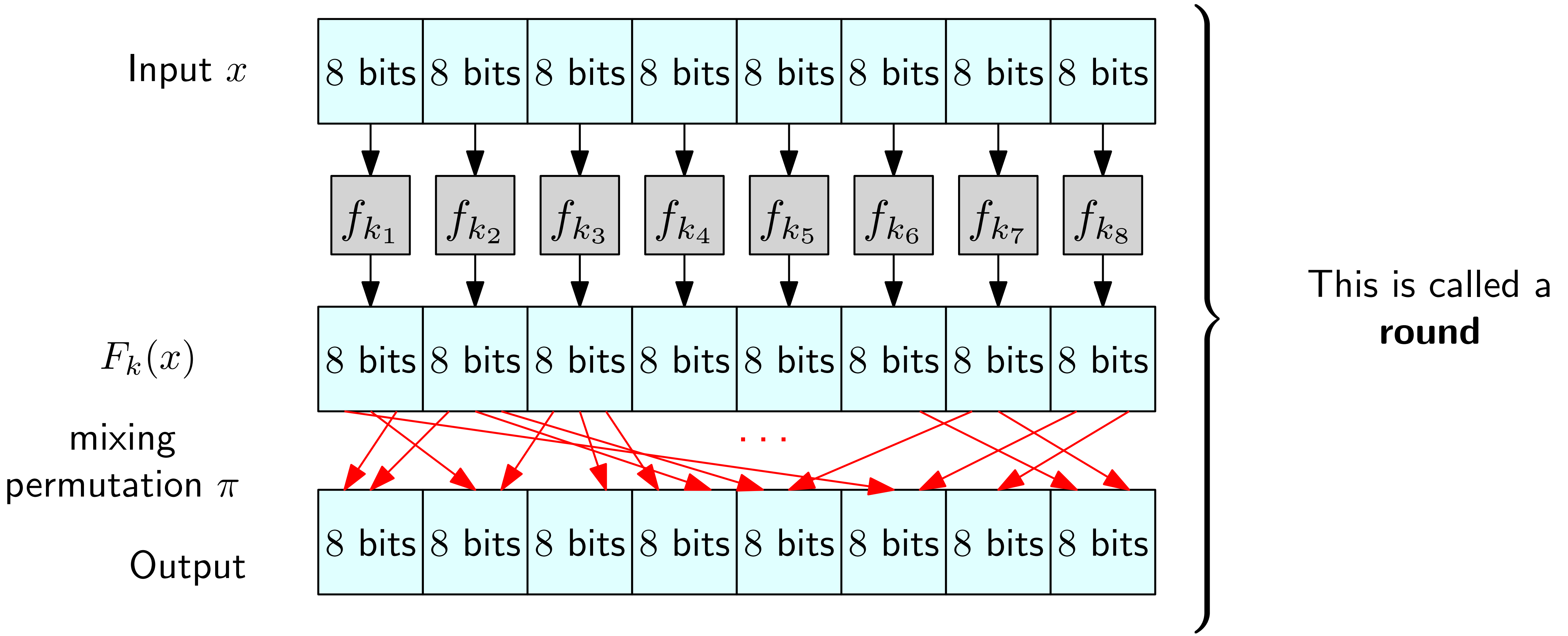
Substitution Permutation Networks



Is the function computed by this SPN a **good** PRP? **No**

- The mixing permutation is fixed. An adversary can always undo the last step!
- We have already argued that $F_k(x)$ is **not** a good PRP.

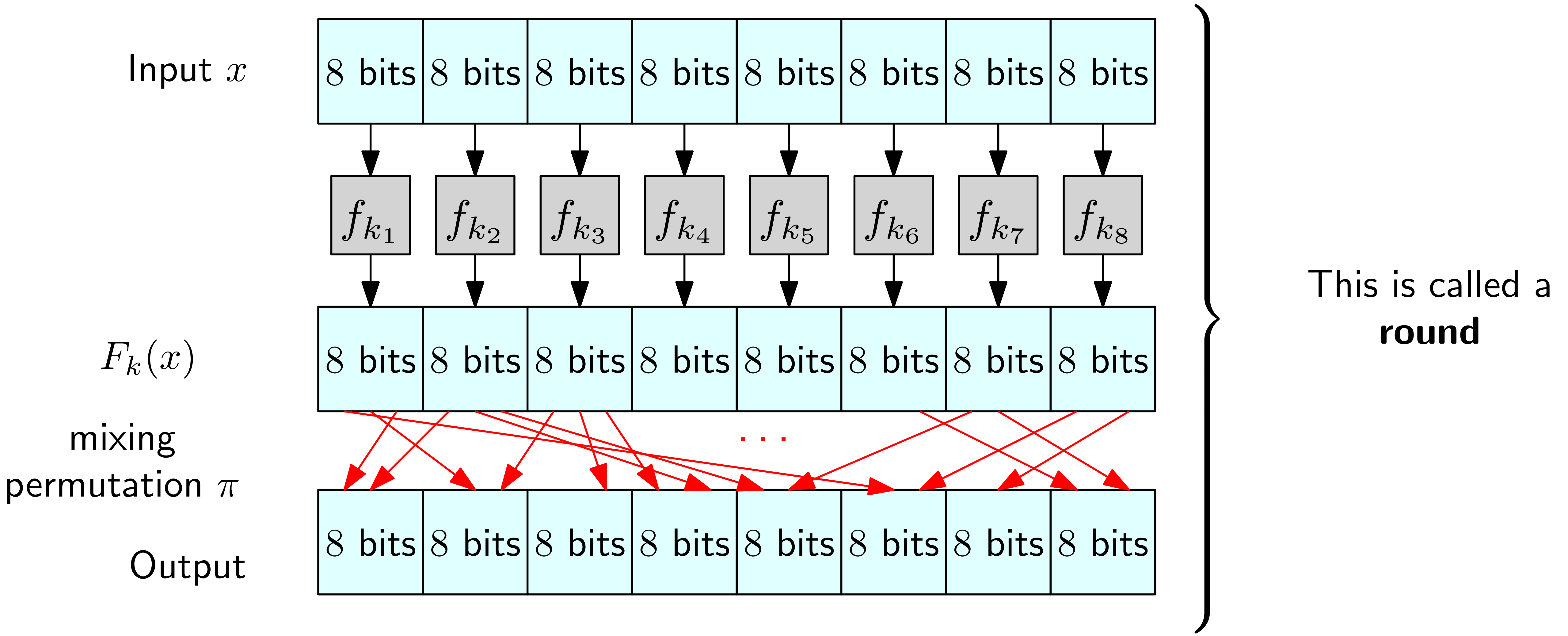
Substitution Permutation Networks



Is the function computed by this SPN a **good** PRP? **No**

- The mixing permutation is fixed. An adversary can always undo the last step!
- We have already argued that $F_k(x)$ is **not** a good PRP.

Substitution Permutation Networks

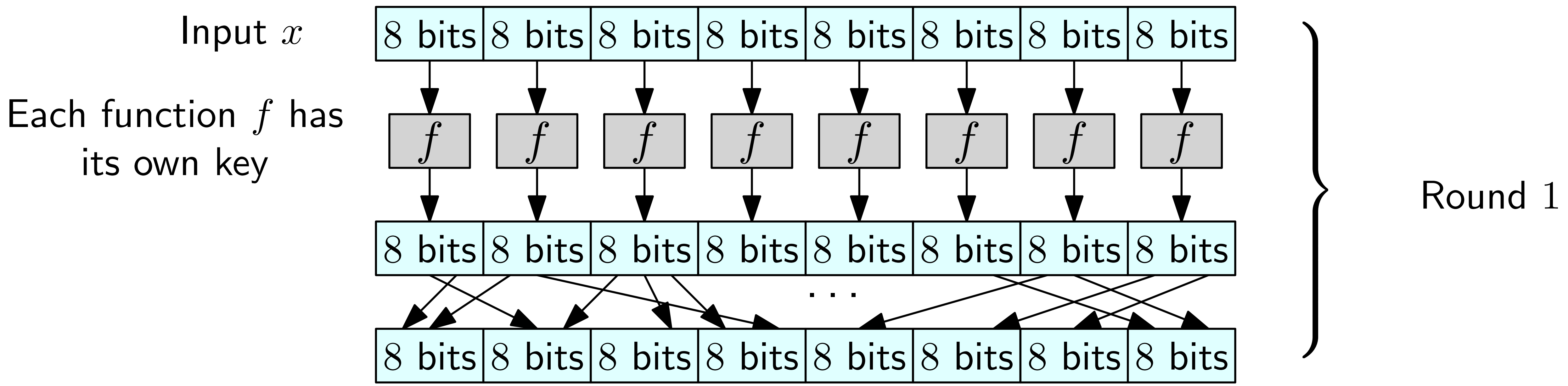


Is the function computed by this SPN a **good** PRP? **No**

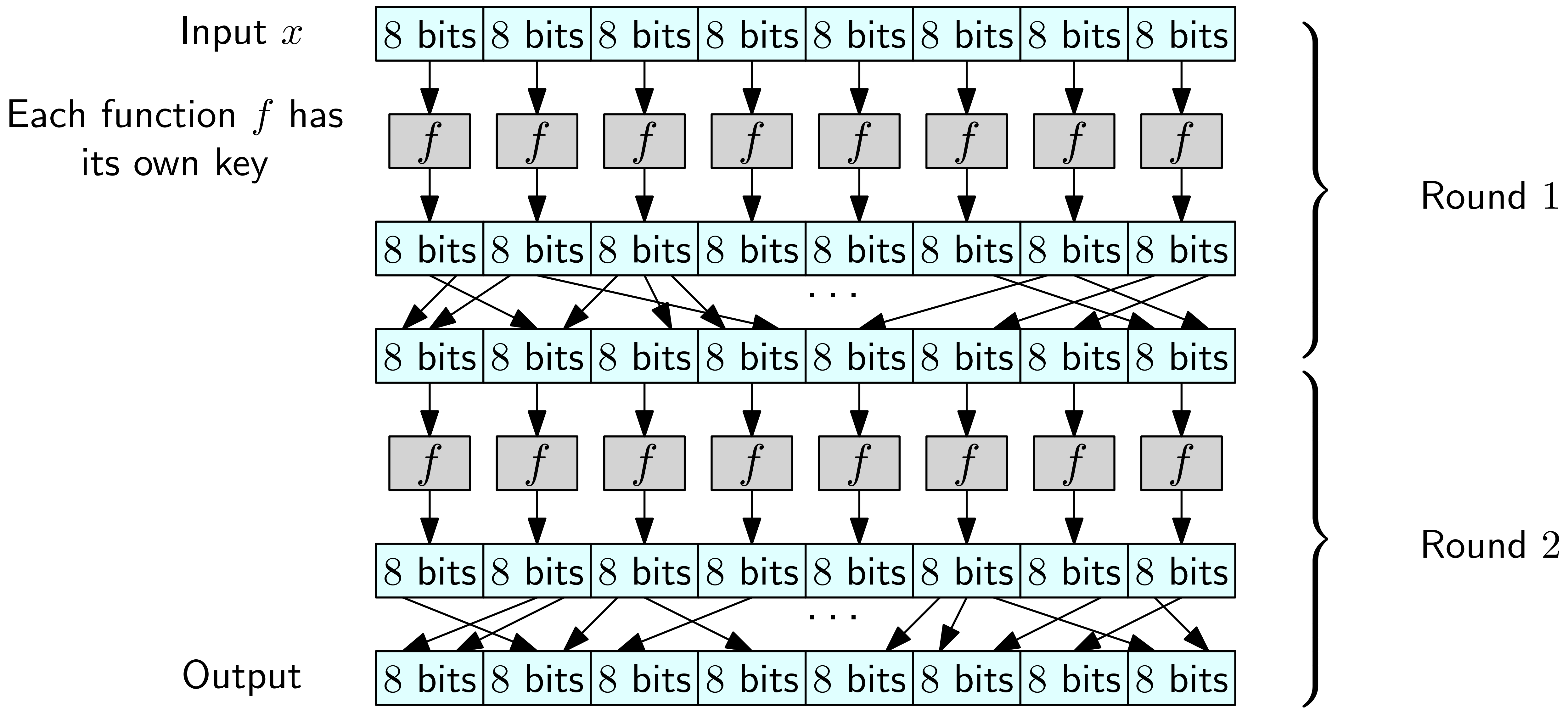
- The mixing permutation is fixed. An adversary can always undo the last step!
- We have already argued that $F_k(x)$ is **not** a good PRP.

What if we do another <u>round</u> with fresh functions f_{k_i}?

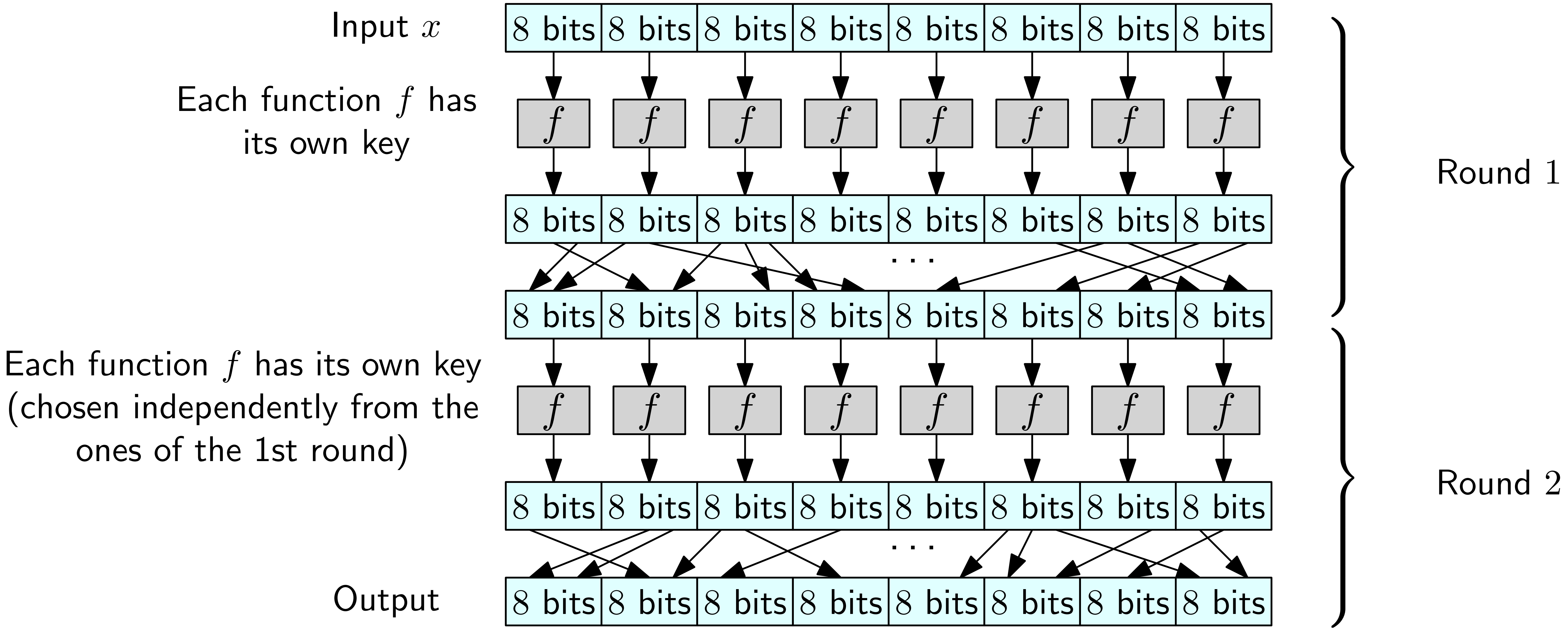
Substitution Permutation Networks



Substitution Permutation Networks



Substitution Permutation Networks



Substitution Permutation Networks

Is the function computed by this SPN a good PRP?

Substitution Permutation Networks

Is the function computed by this SPN a good PRP? **No...**

Substitution Permutation Networks

Is the function computed by this SPN a good PRP? **No...** but it is "better" than before

Substitution Permutation Networks

Is the function computed by this SPN a good PRP? **No...** but it is "better" than before

More rounds!

Substitution Permutation Networks

Is the function computed by this SPN a good PRP? **No...** but it is "better" than before

More rounds!

Observation: the overall permutation remains invertible regardless of the number of rounds

Substitution Permutation Networks

- Using random functions f is unpractical

  The key size would be manageable, but still quite large

Substitution Permutation Networks

- Using random functions f is unpractical

  The key size would be manageable, but still quite large

- We restrict ourselves to functions f that have a particular form:

$$f_{k,i}(x) = S_i(k_i \oplus x_i)$$

Substitution Permutation Networks

- Using random functions f is unpractical

  The key size would be manageable, but still quite large

- We restrict ourselves to functions f that have a particular form:

$$f_{k,i}(x) = S_i(k_i \oplus x_i)$$

- The XOR-ing operation is called **key mixing**

Substitution Permutation Networks

- Using random functions f is unpractical

  The key size would be manageable, but still quite large

- We restrict ourselves to functions f that have a particular form:

$$f_{k,i}(x) = S_i(k_i \oplus x_i)$$

- The XOR-ing operation is called **key mixing**
- The functions S_i are called **S-boxes** (from substitution boxes)

Substitution Permutation Networks

- Using random functions f is unpractical

  The key size would be manageable, but still quite large

- We restrict ourselves to functions f that have a particular form:

$$f_{k,i}(x) = S_i(k_i \oplus x_i)$$

- The XOR-ing operation is called **key mixing**
- The functions S_i are called **S-boxes** (from substitution boxes)
- The key $k = k_1 \parallel k_2 \parallel k_3 \parallel \ldots$ is called **sub-key** or **round key**

Substitution Permutation Networks

- Using random functions f is unpractical

  The key size would be manageable, but still quite large

- We restrict ourselves to functions f that have a particular form:

$$f_{k,i}(x) = S_i(k_i \oplus x_i)$$

- The XOR-ing operation is called **key mixing**
- The functions S_i are called **S-boxes** (from substitution boxes)
- The key $k = k_1 \parallel k_2 \parallel k_3 \parallel \dots$ is called **sub-key** or **round key**
- Different rounds use different round keys

Substitution Permutation Networks

- Using random functions f is unpractical

  The key size would be manageable, but still quite large

- We restrict ourselves to functions f that have a particular form:

$$f_{k,i}(x) = S_i(k_i \oplus x_i)$$

- The XOR-ing operation is called **key mixing**
- The functions S_i are called **S-boxes** (from substitution boxes)
- The key $k = k_1 \parallel k_2 \parallel k_3 \parallel \ldots$ is called **sub-key** or **round key**
- Different rounds use different round keys
- The key of the whole block cipher is called the **master key**

Substitution Permutation Networks

- Using random functions f is unpractical

  The key size would be manageable, but still quite large

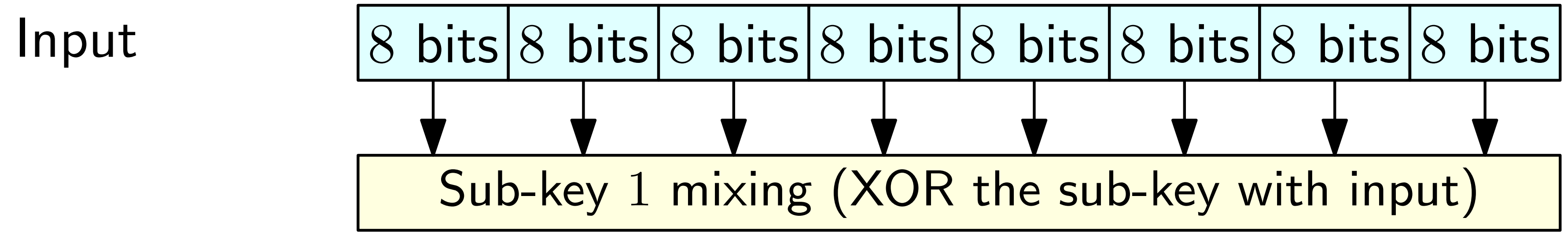
- We restrict ourselves to functions f that have a particular form:

$$f_{k,i}(x) = S_i(k_i \oplus x_i)$$

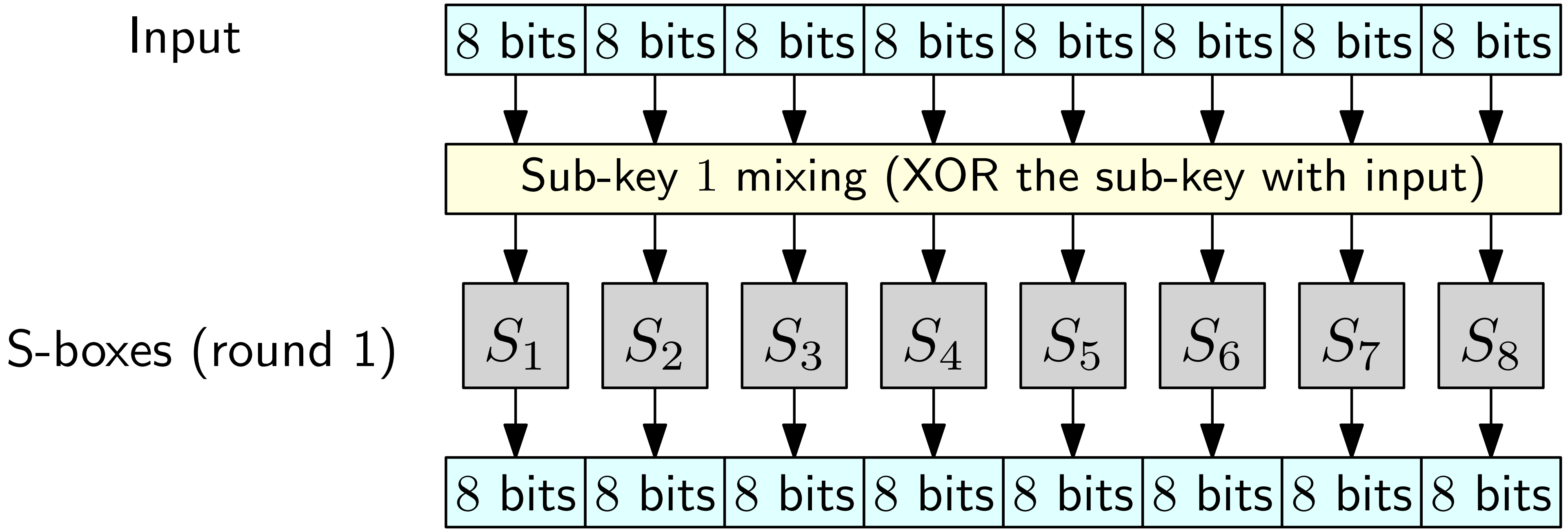
- The XOR-ing operation is called **key mixing**
- The functions S_i are called **S-boxes** (from substitution boxes)
- The key $k = k_1 \| k_2 \| k_3 \| \ldots$ is called **sub-key** or **round key**
- Different rounds use different round keys
- The key of the whole block cipher is called the **master key**
- The round keys are derived from the master key according to a **key schedule**

Input	8 bits	8 bits	8 bits	8 bits	8 bits	8 bits	8 bits	8 bits

Sample structure of a 2-round block cipher

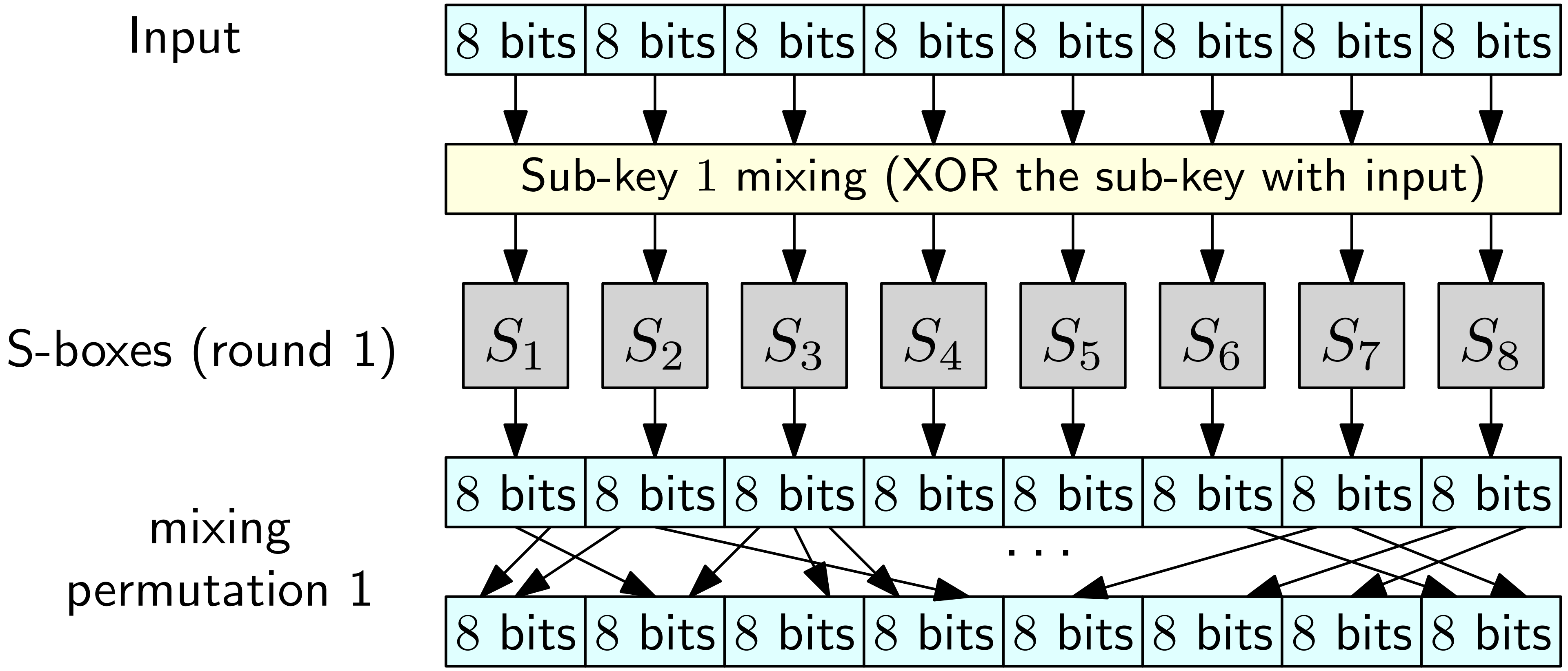


Sample structure of a
2-round block cipher

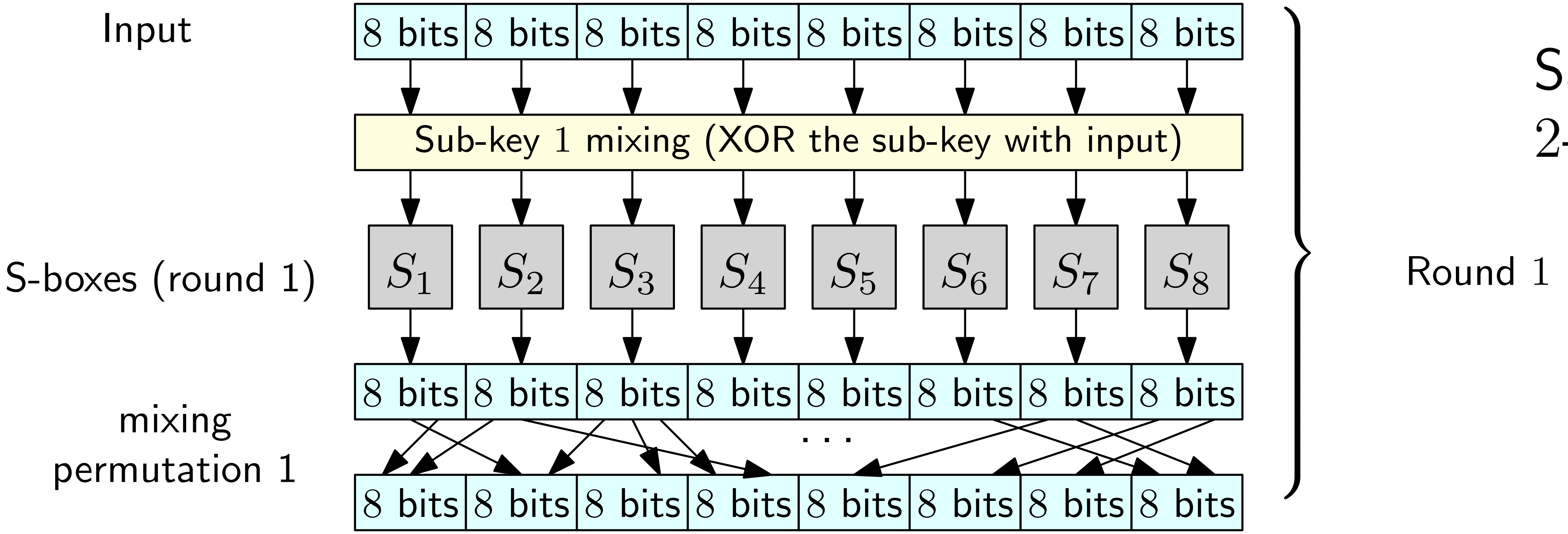


Sample structure of a 2-round block cipher

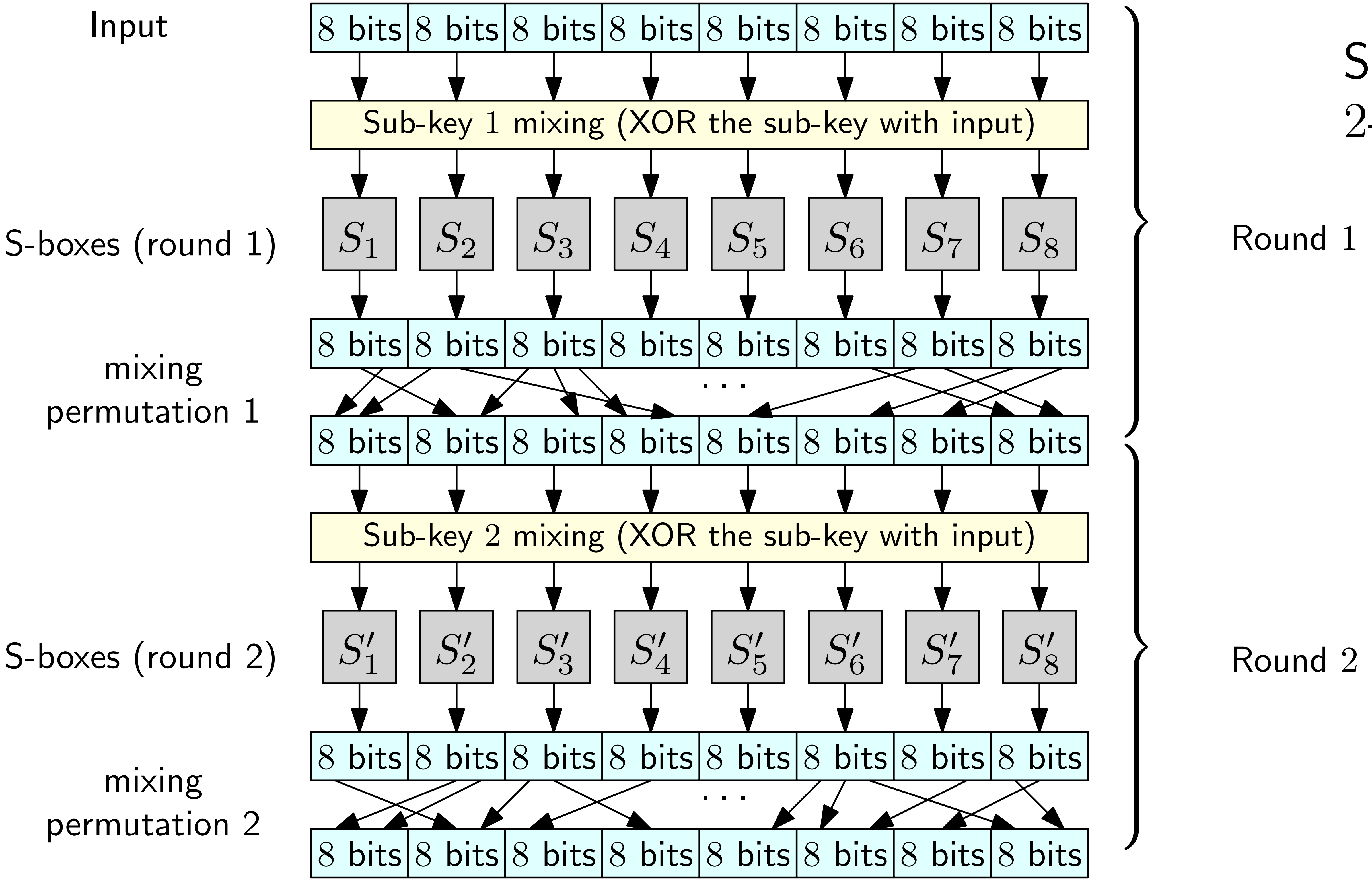
Sample structure of a 2-round block cipher



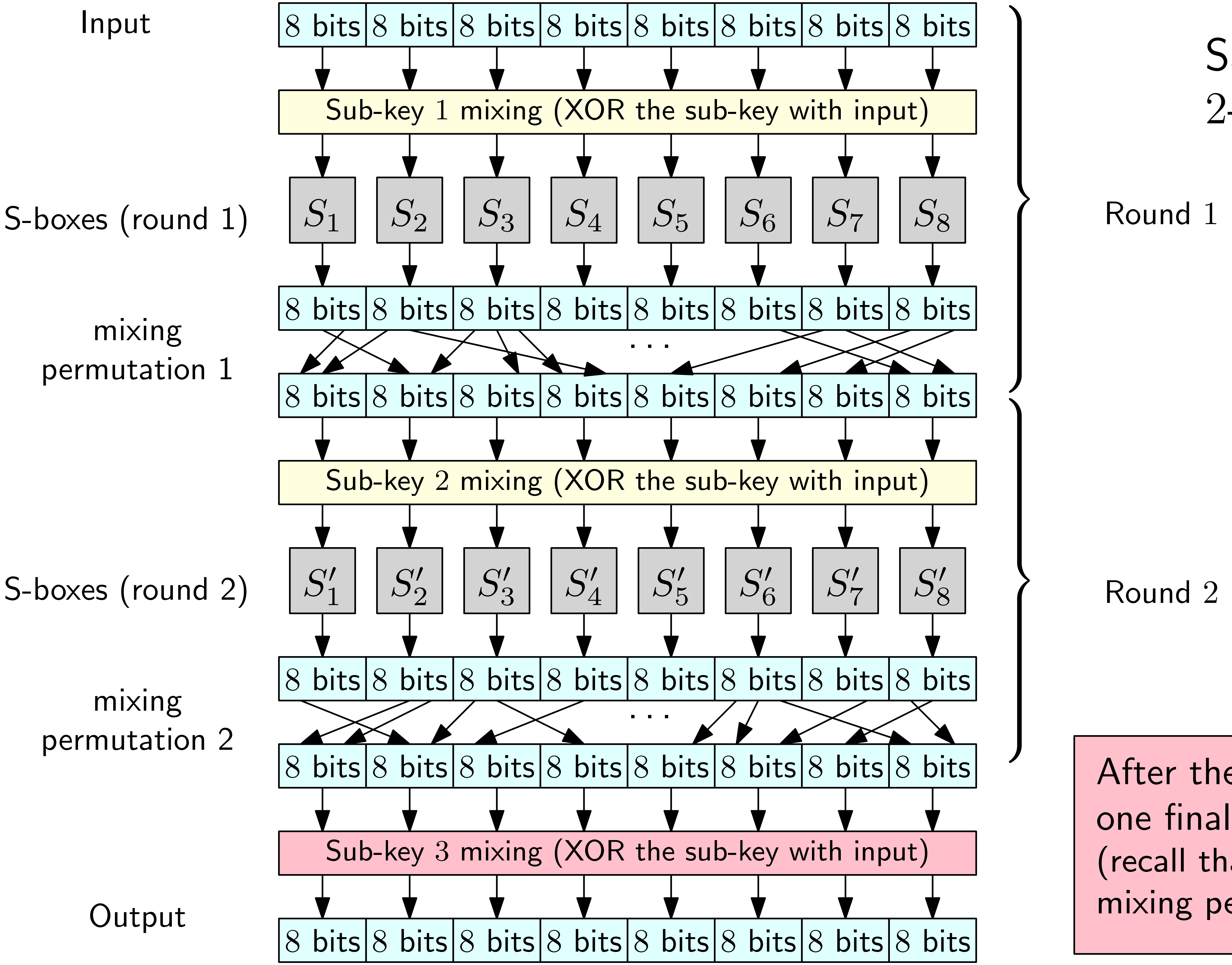
Sample structure of a 2-round block cipher



Sample structure of a 2-round block cipher



Sample structure of a 2-round block cipher



After the last round, we perform one final key mixing step (recall that it is useless to apply a mixing permutation as the last step)

The Avalanche Effect

We want to design the S-boxes and the mixing permutation to achieve the **avalanche effect**

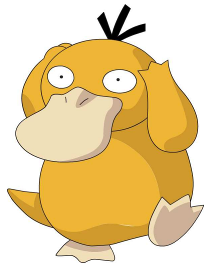
- Even a small difference in the input should eventually (over multiple rounds) propagate to the entire output

The Avalanche Effect

We want to design the S-boxes and the mixing permutation to achieve the **avalanche effect**

- Even a small difference in the input should eventually (over multiple rounds) propagate to the entire output

For S-boxes:



- Any 1-bit change in the input should cause ≥ 2 bits to change in the output
- This adds confusion

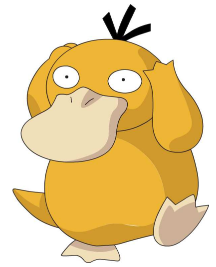
The Avalanche Effect

We want to design the S-boxes and the mixing permutation to achieve the **avalanche effect**

- Even a small difference in the input should eventually (over multiple rounds) propagate to the entire output

For S-boxes:

- Any 1-bit change in the input should cause ≥ 2 bits to change in the output
- This adds confusion

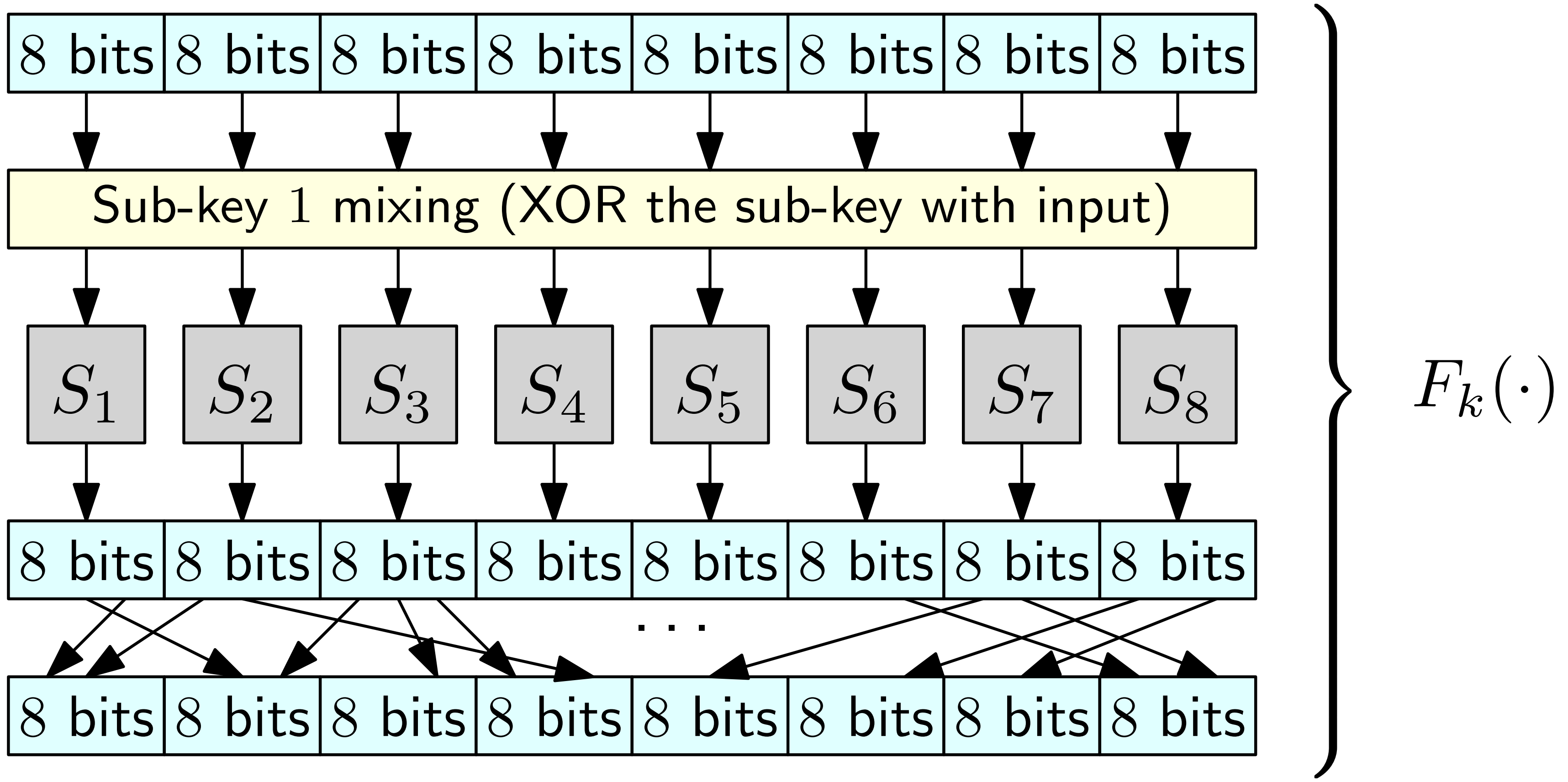


For the mixing permutation:

- A bit output from a S-box should be fed into a different S-box into the next round
- This adds diffusion

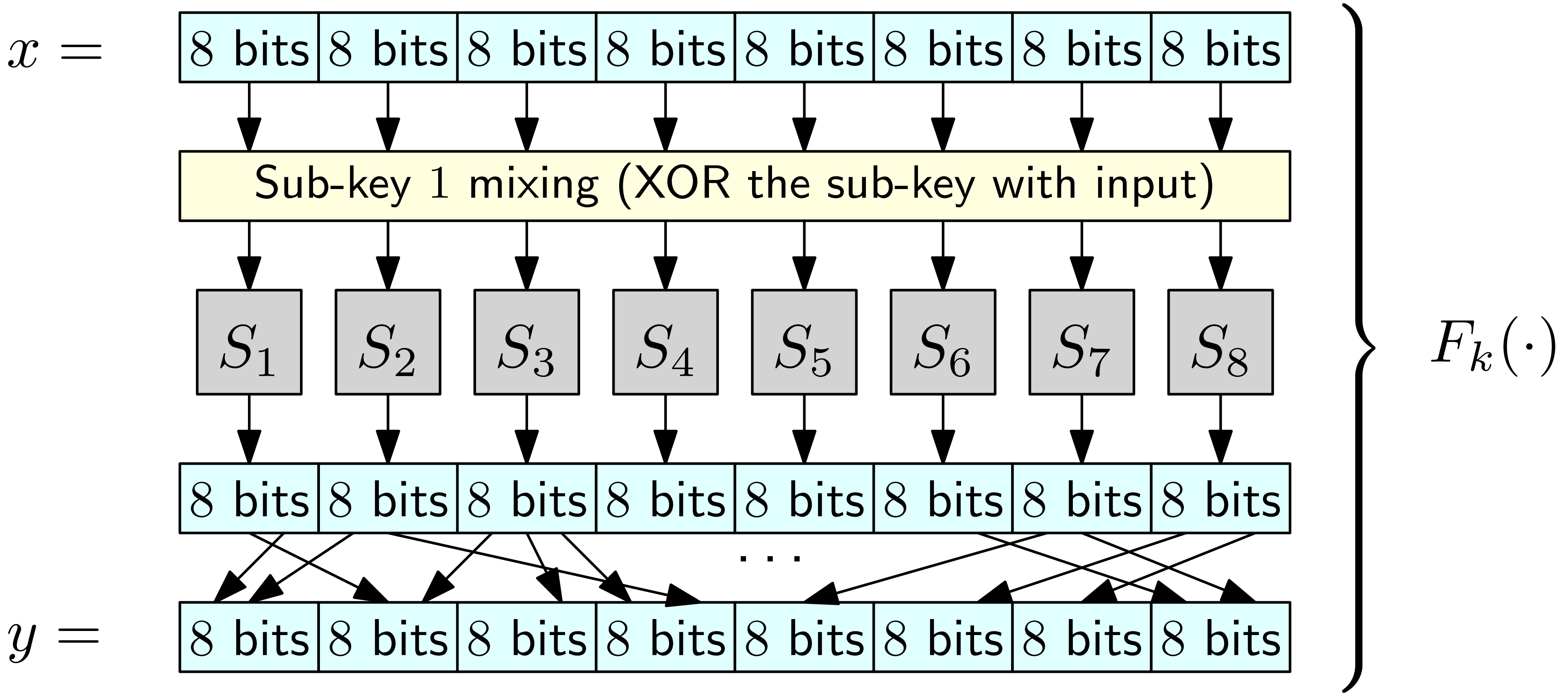
Key recovery attack against a simplified 1-round SPN

Simple case: 1-round SPN and no final key mixing step



Key recovery attack against a simplified 1-round SPN

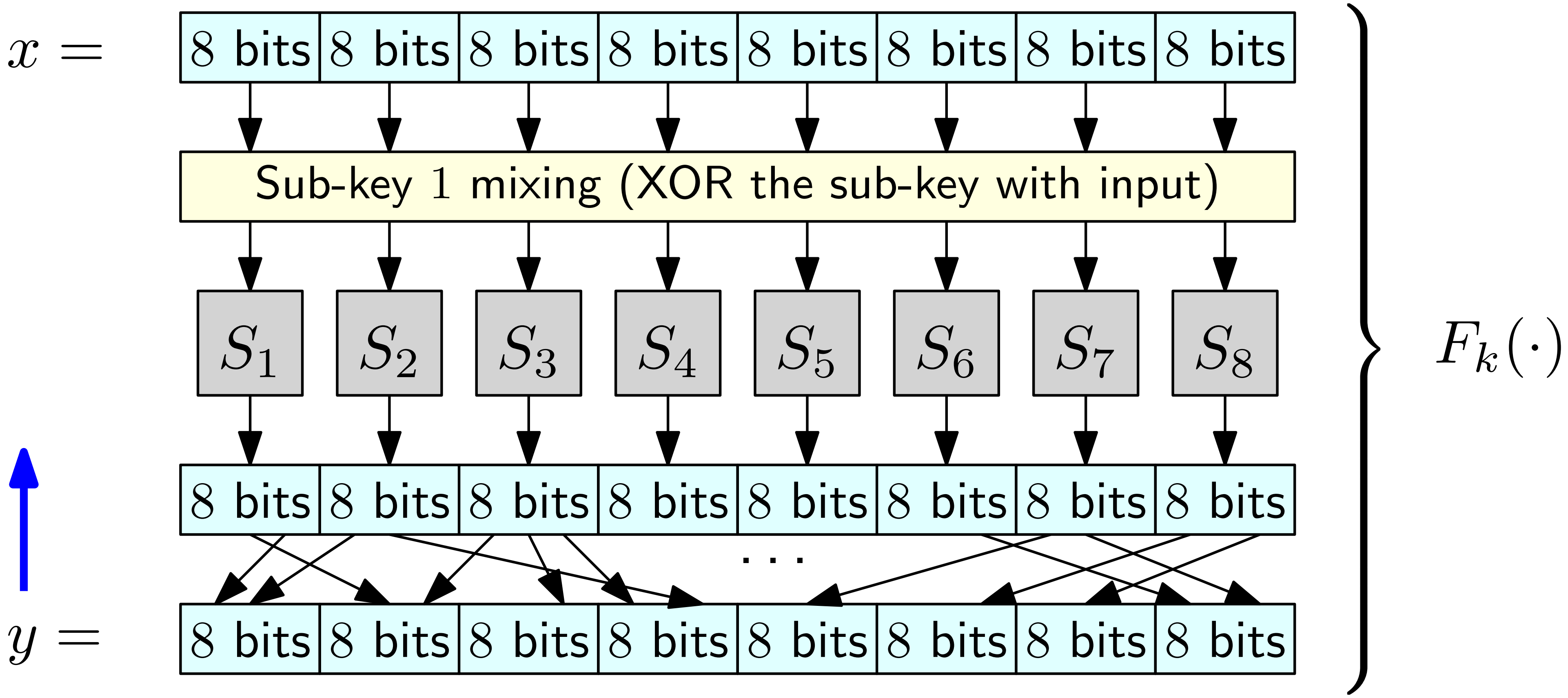
Simple case: 1-round SPN and no final key mixing step



The adversary can recover the key from a single input-output pair $x, y = F_k(x)$ **How?**

Key recovery attack against a simplified 1-round SPN

Simple case: 1-round SPN and no final key mixing step

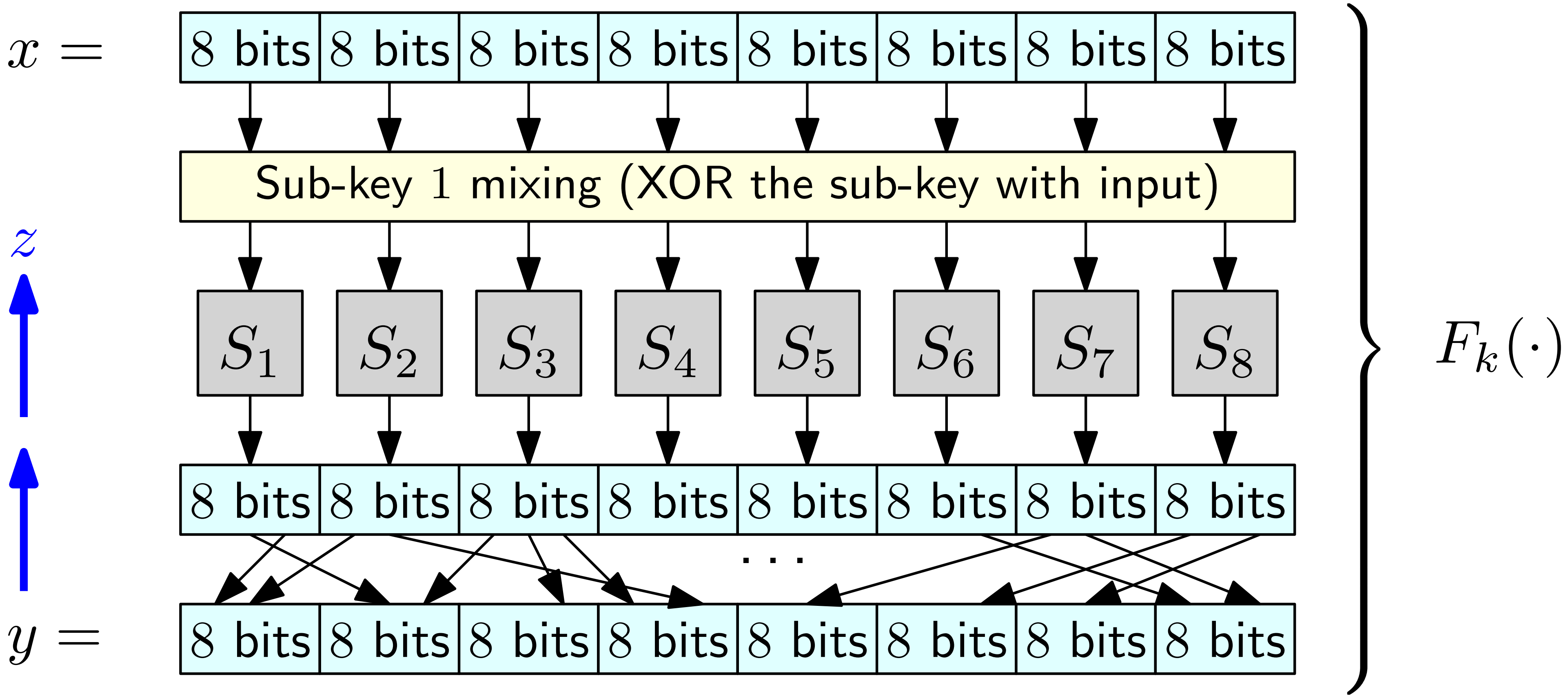


The adversary can recover the key from a single input-output pair $x, y = F_k(x)$ **How?**

- Invert the mixing permutation (it is fixed and known to the attacker)

Key recovery attack against a simplified 1-round SPN

Simple case: 1-round SPN and no final key mixing step

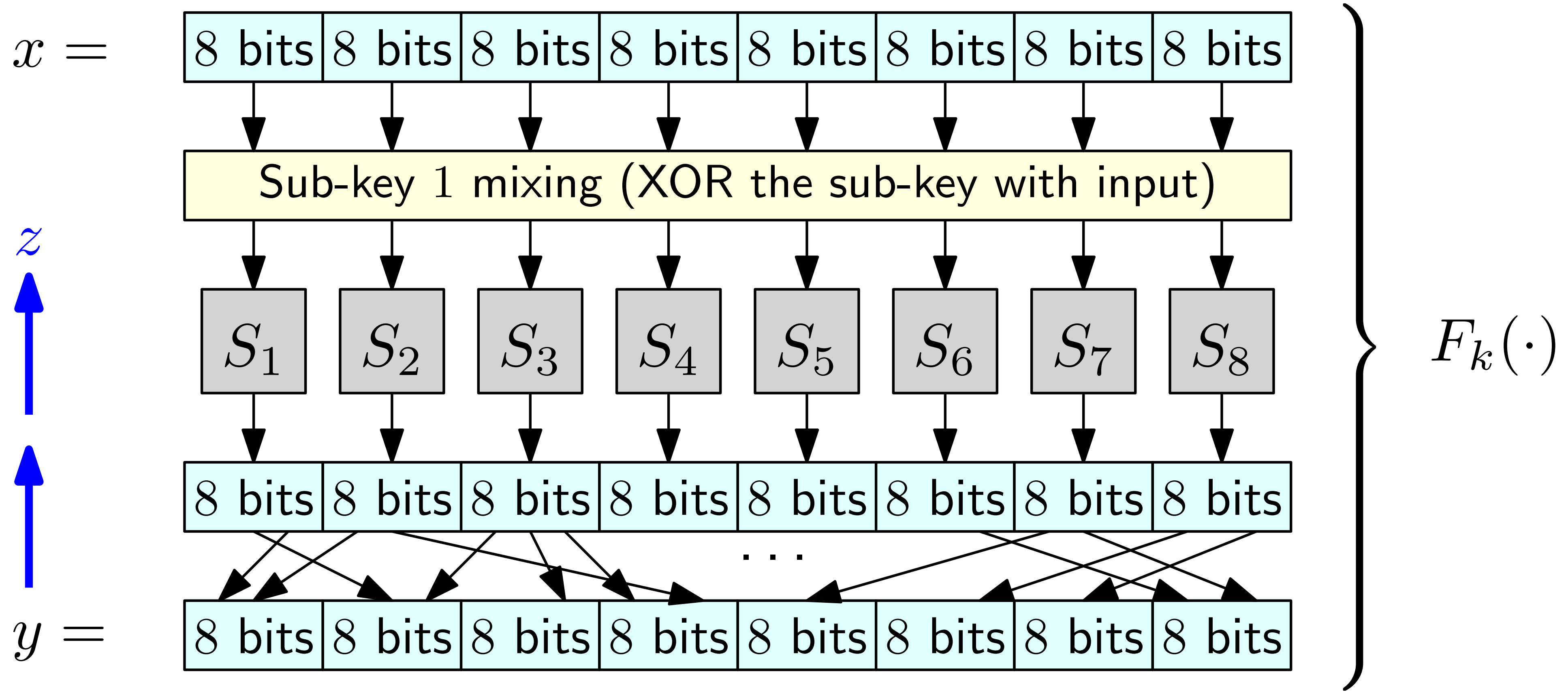


The adversary can recover the key from a single input-output pair $x, y = F_k(x)$ **How?**

- Invert the mixing permutation (it is fixed and known to the attacker)
- Invert the S-boxes, the computed value will be exactly $z = x \oplus k$

Key recovery attack against a simplified 1-round SPN

Simple case: 1-round SPN and no final key mixing step

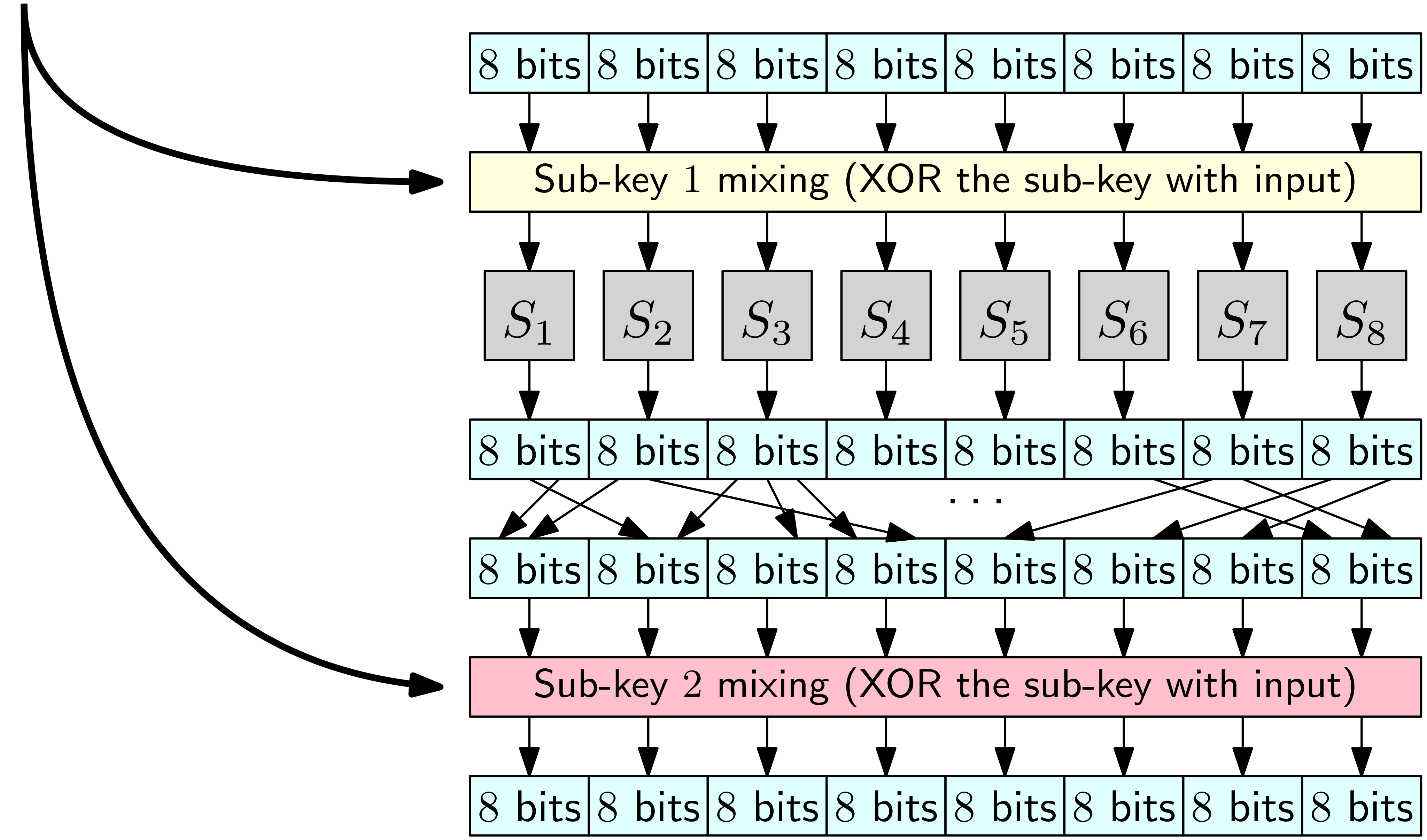


The adversary can recover the key from a single input-output pair $x, y = F_k(x)$ **How?**

- Invert the mixing permutation (it is fixed and known to the attacker)
- Invert the S-boxes, the computed value will be exactly $z = x \oplus k$
- The (round and master) key is $k = z \oplus x = (x \oplus k) \oplus x$

Key recovery attack against a full 1-round SPN

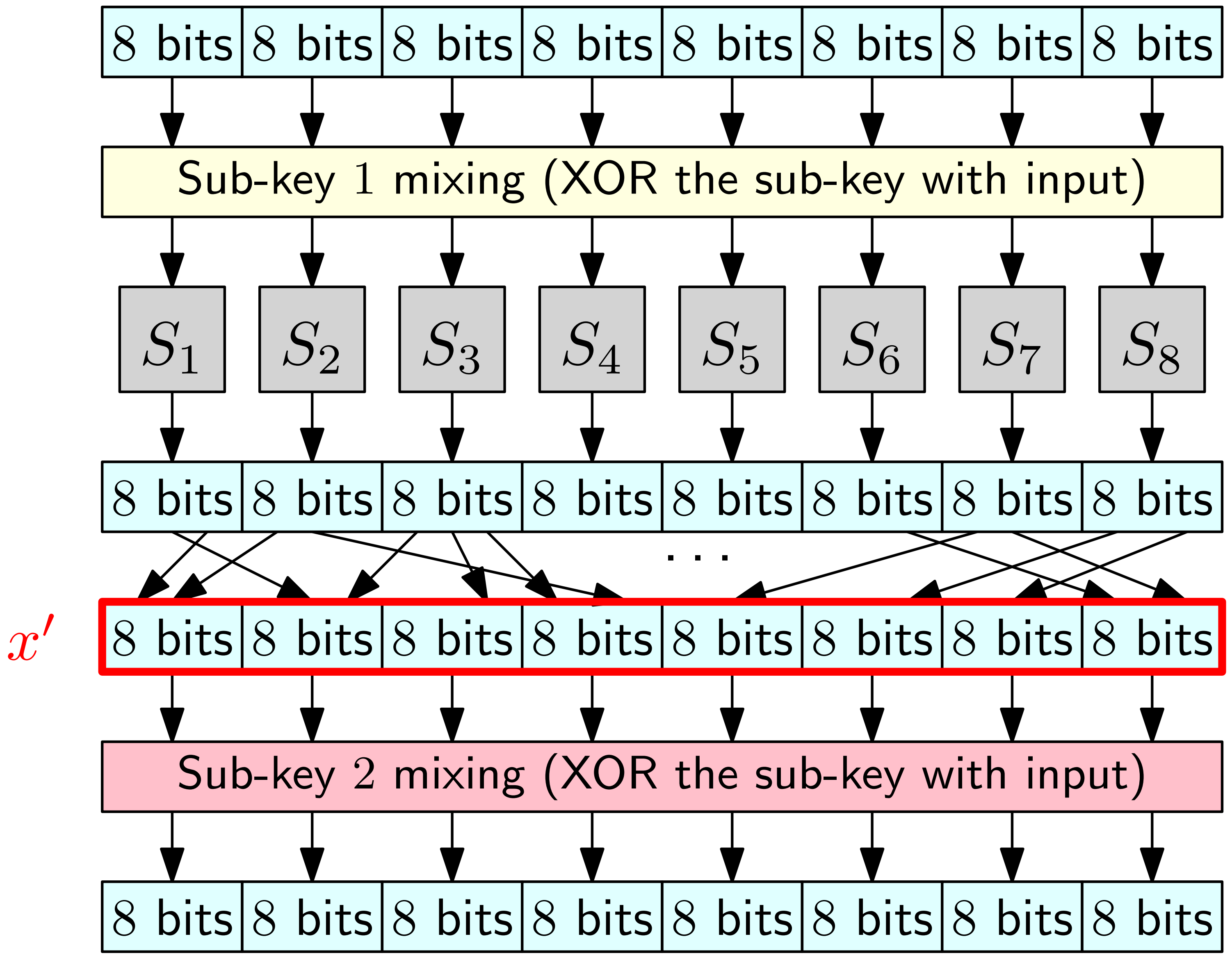
Consider now a **full** 1-round SPN (with the final key mixing step), in which the master key is just the concatenation of two independent sub-keys



Key recovery attack against a full 1-round SPN

Consider now a **full** 1-round SPN (with the final key mixing step), in which the master key is just the concatenation of two independent sub-keys

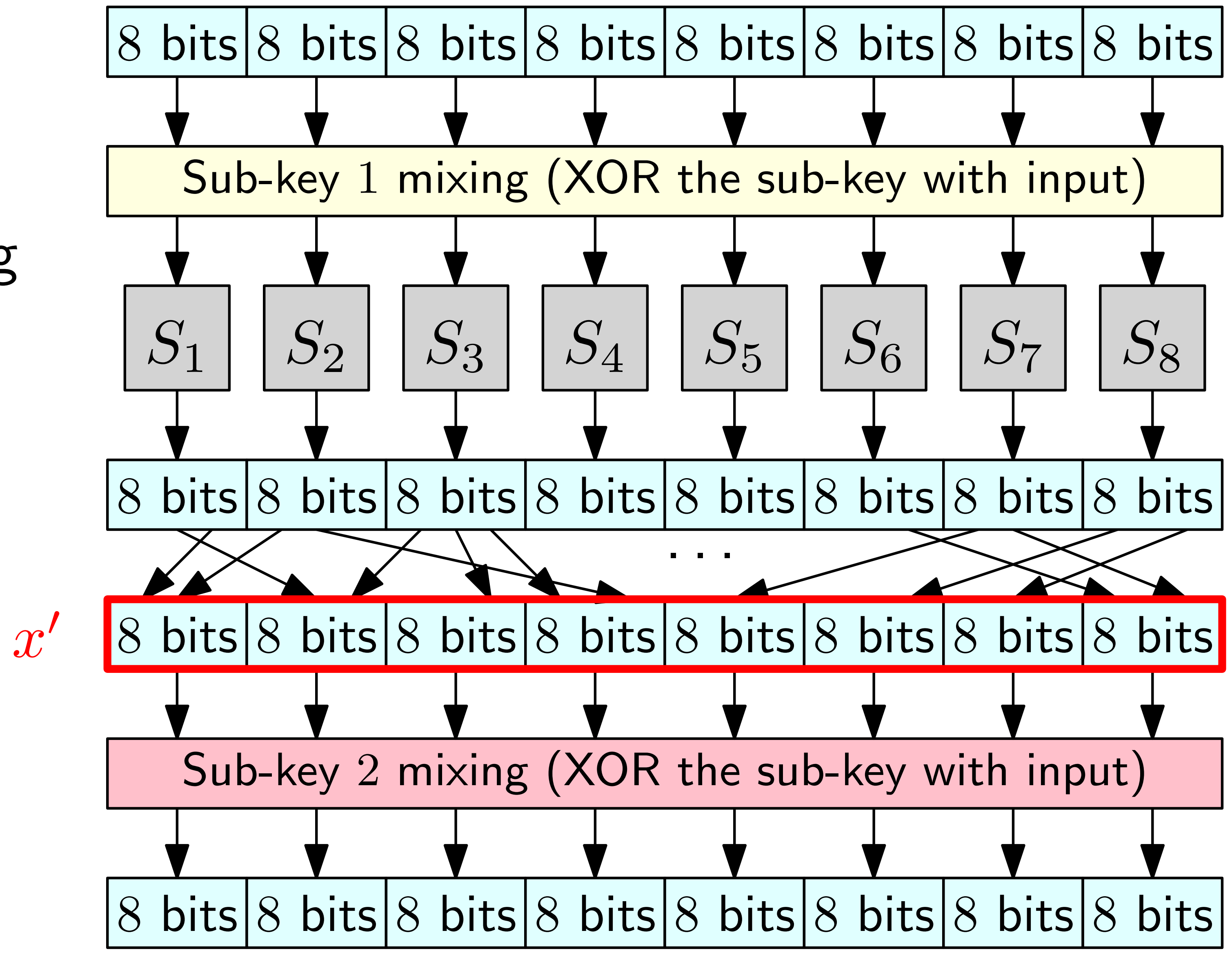
- Try all possible 1st sub-keys. For each of them use the input x to determine the input x' to the final mixing step



Key recovery attack against a full 1-round SPN

Consider now a **full** 1-round SPN (with the final key mixing step), in which the master key is just the concatenation of two independent sub-keys

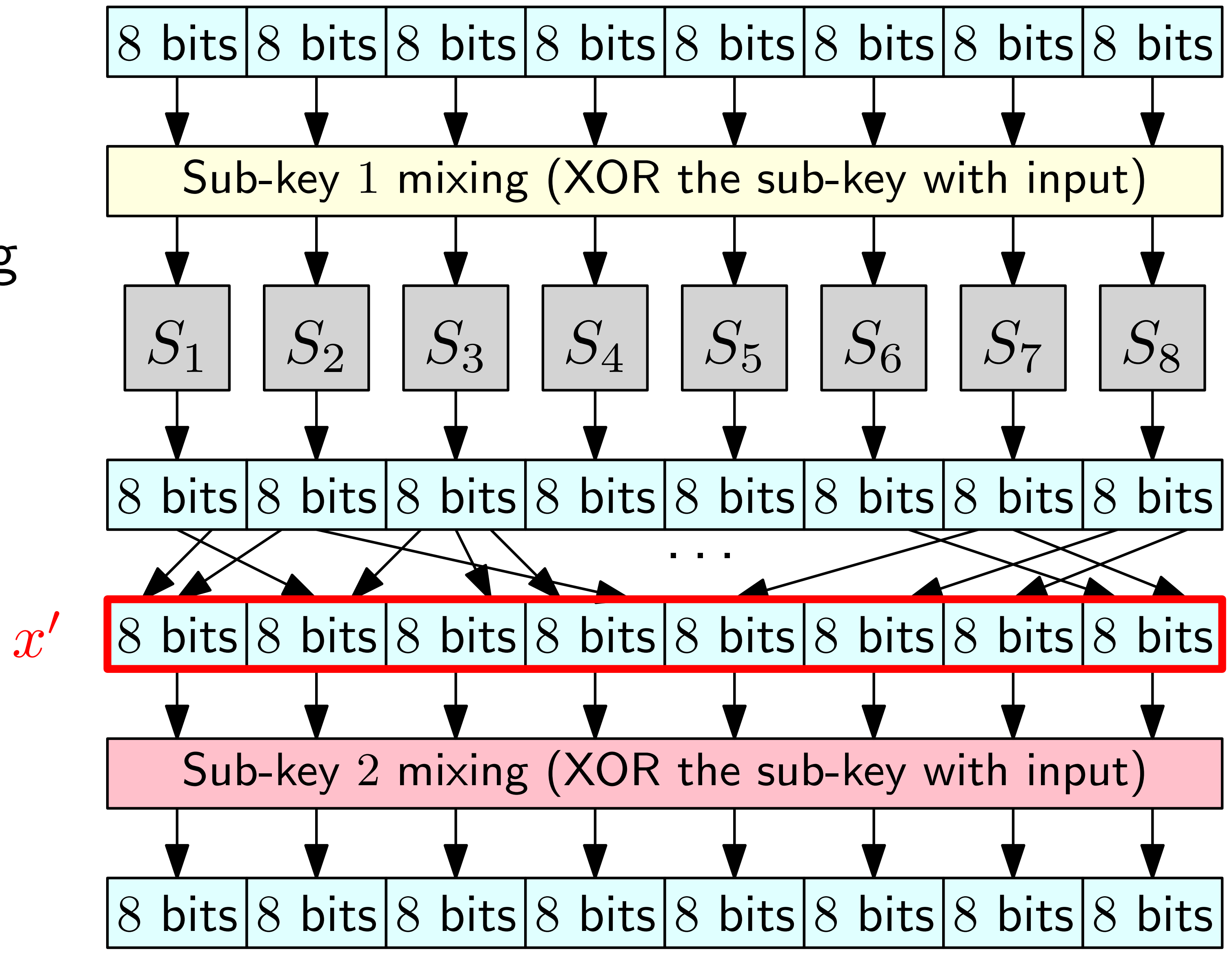
- Try all possible 1st sub-keys. For each of them use the input x to determine the input x' to the final mixing step
- Use the previous strategy to recover the 2nd mixing sub-key from x' and y



Key recovery attack against a full 1-round SPN

Consider now a **full** 1-round SPN (with the final key mixing step), in which the master key is just the concatenation of two independent sub-keys

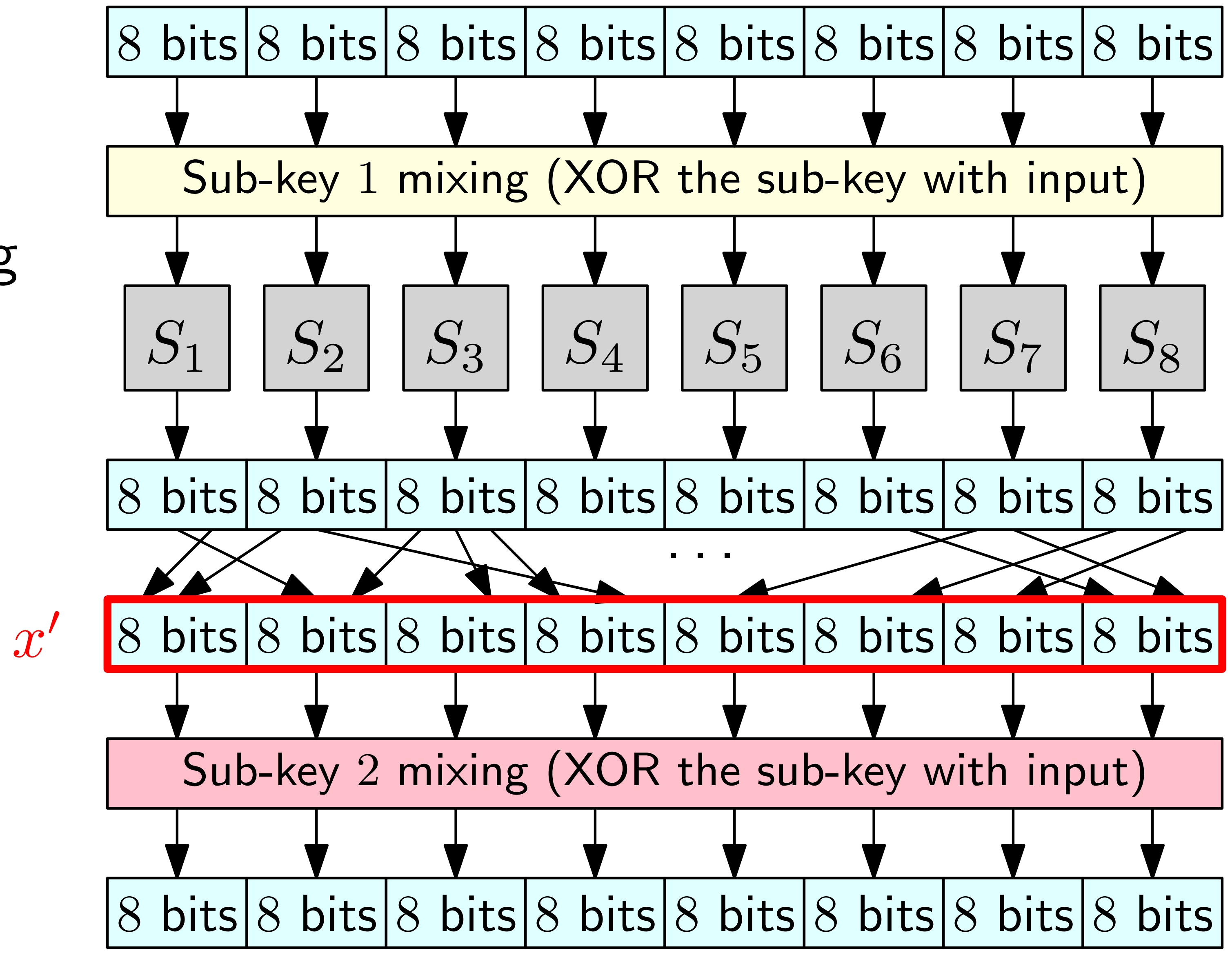
- Try all possible 1st sub-keys. For each of them use the input x to determine the input x' to the final mixing step
- Use the previous strategy to recover the 2nd mixing sub-key from x' and y
- This provides 2^n candidate pairs of keys. Use multiple input-output pairs to eliminate the wrong pairs



Key recovery attack against a full 1-round SPN

Consider now a **full** 1-round SPN (with the final key mixing step), in which the master key is just the concatenation of two independent sub-keys

- Try all possible 1st sub-keys. For each of them use the input x to determine the input x' to the final mixing step
- Use the previous strategy to recover the 2nd mixing sub-key from x' and y
- This provides 2^n candidate pairs of keys. Use multiple input-output pairs to eliminate the wrong pairs
- Time: $\approx 2^n = \sqrt{2^N}$ to recover the master key of length $N = 2n$

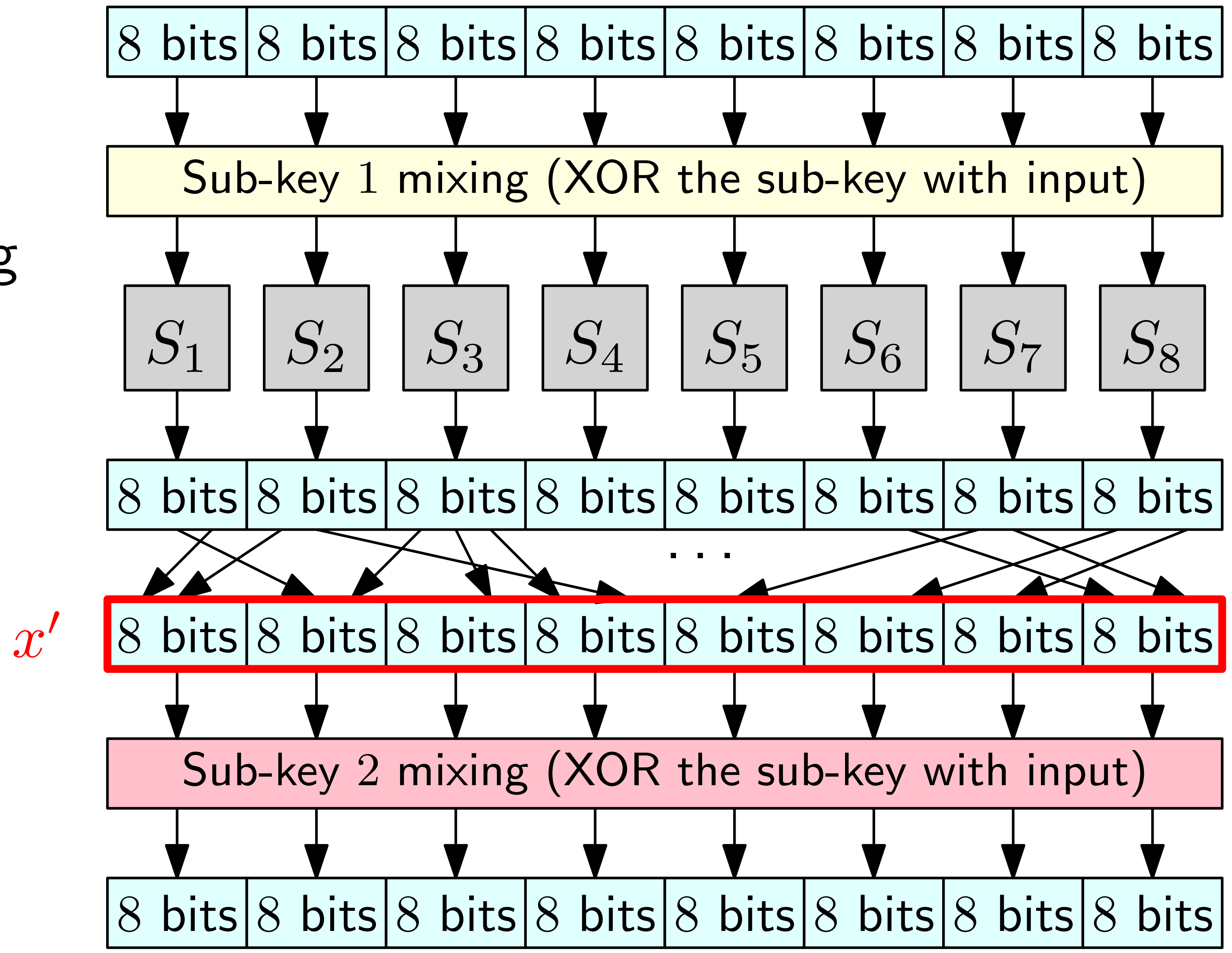


Key recovery attack against a full 1-round SPN

Consider now a **full** 1-round SPN (with the final key mixing step), in which the master key is just the concatenation of two independent sub-keys

- Try all possible 1st sub-keys. For each of them use the input x to determine the input x' to the final mixing step
- Use the previous strategy to recover the 2nd mixing sub-key from x' and y
- This provides 2^n candidate pairs of keys. Use multiple input-output pairs to eliminate the wrong pairs
- Time: $\approx 2^n = \sqrt{2^N}$ to recover the master key of length $N = 2n$

— Altough this is not polynomially bounded, we would like all (known) attacks to take time $\approx 2^N$



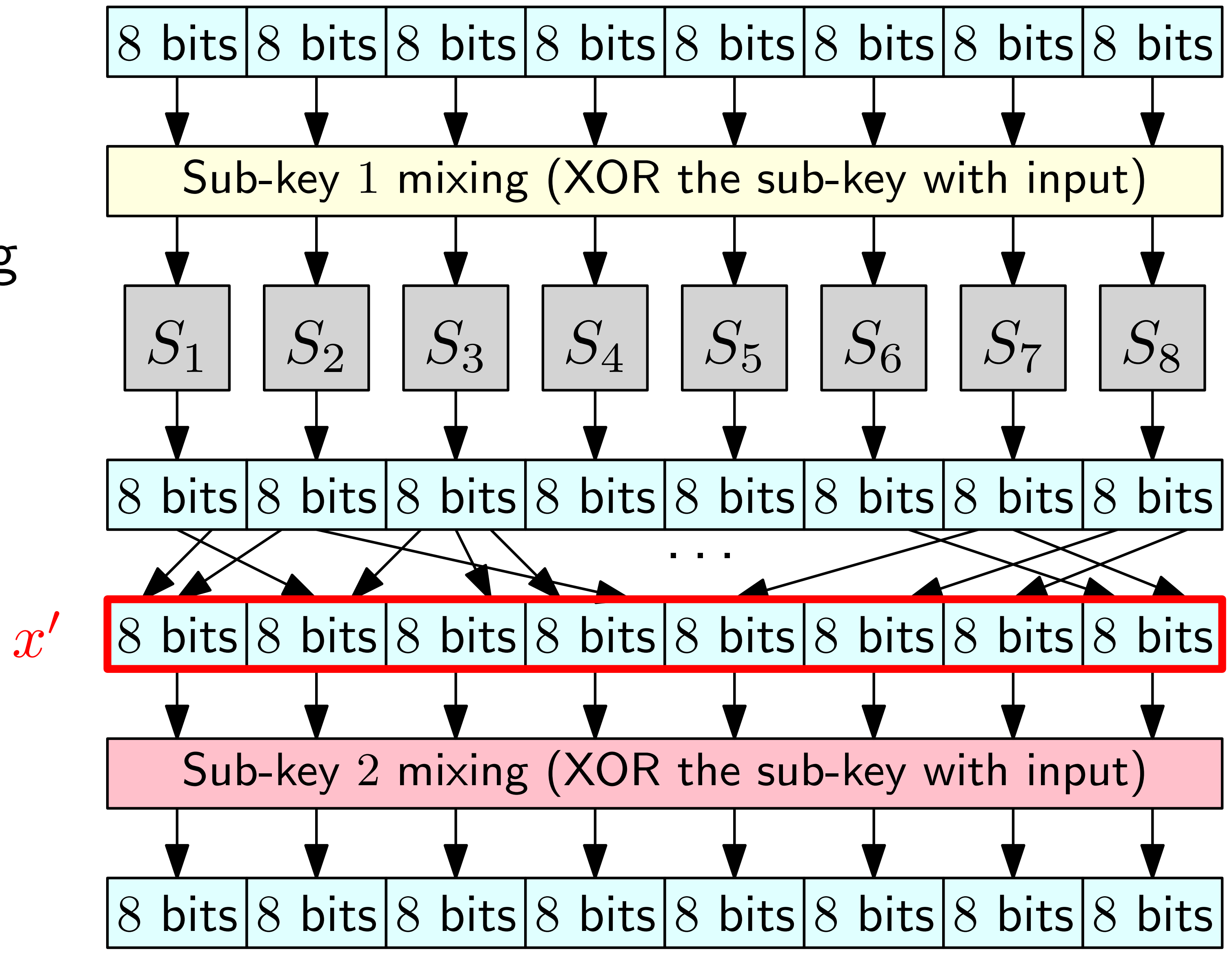
Key recovery attack against a full 1-round SPN

Consider now a **full** 1-round SPN (with the final key mixing step), in which the master key is just the concatenation of two independent sub-keys

- Try all possible 1st sub-keys. For each of them use the input x to determine the input x' to the final mixing step
- Use the previous strategy to recover the 2nd mixing sub-key from x' and y
- This provides 2^n candidate pairs of keys. Use multiple input-output pairs to eliminate the wrong pairs
- Time: $\approx 2^n = \sqrt{2^N}$ to recover the master key of length $N = 2n$

— Altough this is not polynomially bounded, we would like all (known) attacks to take time $\approx 2^N$

— Attacks faster than bruteforce might be symptoms of more fundamental weaknesses



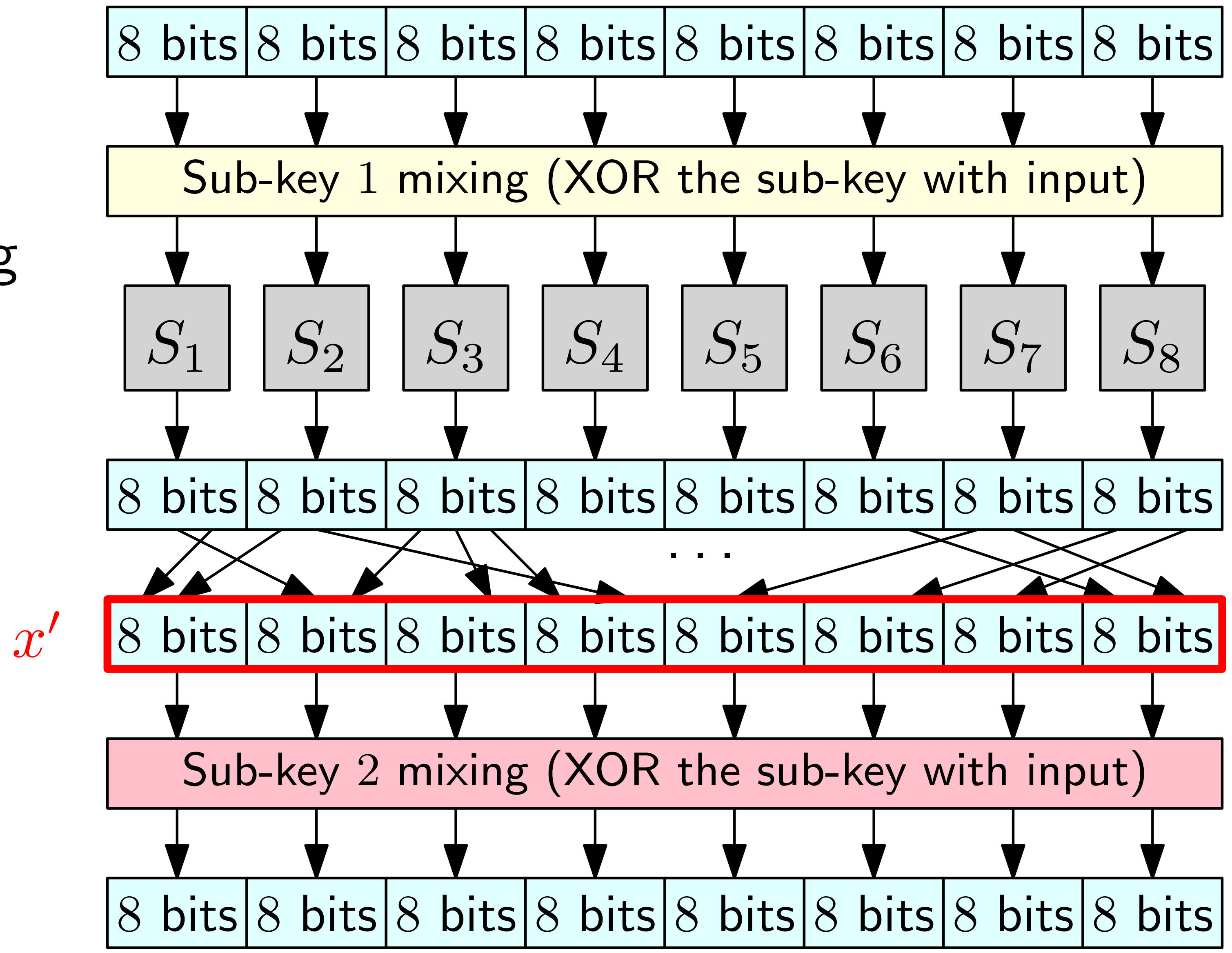
Key recovery attack against a full 1-round SPN

Consider now a **full** 1-round SPN (with the final key mixing step), in which the master key is just the concatenation of two independent sub-keys

- Try all possible 1st sub-keys. For each of them use the input x to determine the input x' to the final mixing step
- Use the previous strategy to recover the 2nd mixing sub-key from x' and y
- This provides 2^n candidate pairs of keys. Use multiple input-output pairs to eliminate the wrong pairs
- Time: $\approx 2^n = \sqrt{2^N}$ to recover the master key of length $N = 2n$

— Altough this is not polynomially bounded, we would like all (known) attacks to take time $\approx 2^N$

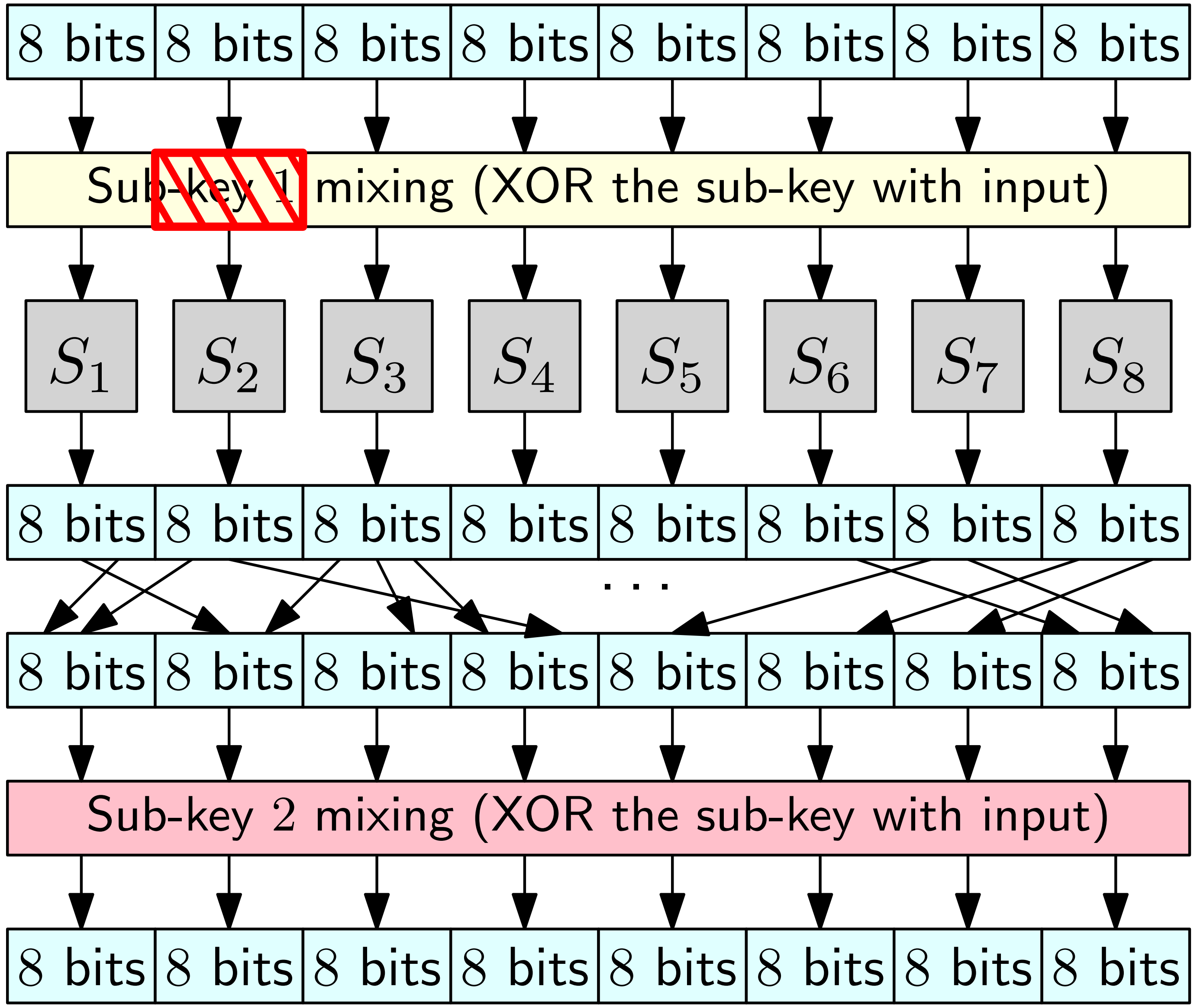
— Attacks faster than bruteforce might be symptoms of more fundamental weaknesses



Indeed... we can design a better attack!

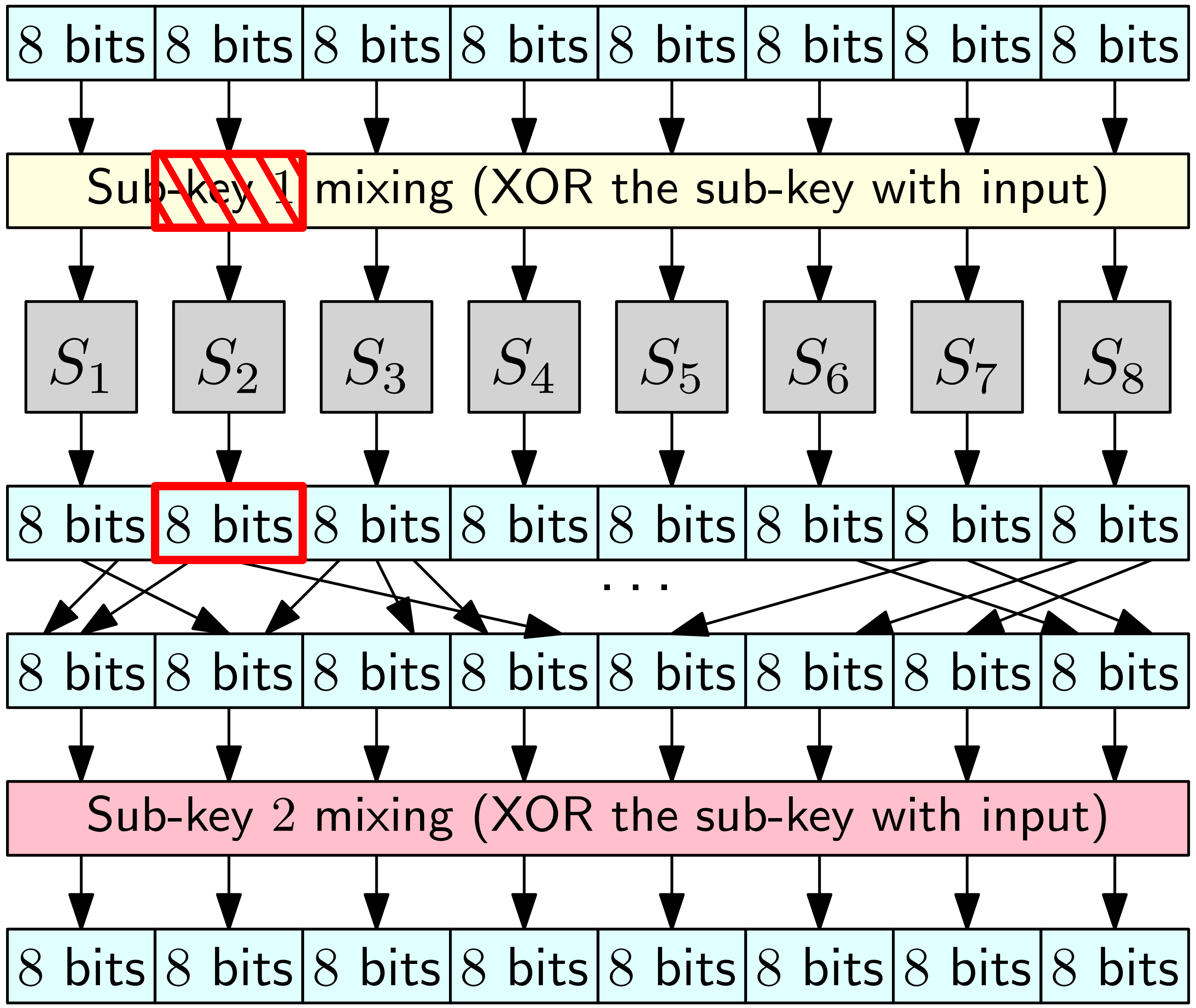
A better key recovery attack against a full 1-round SPN

- Guess only the part of 1st mixing sub-key that contributes to the input of some S-box



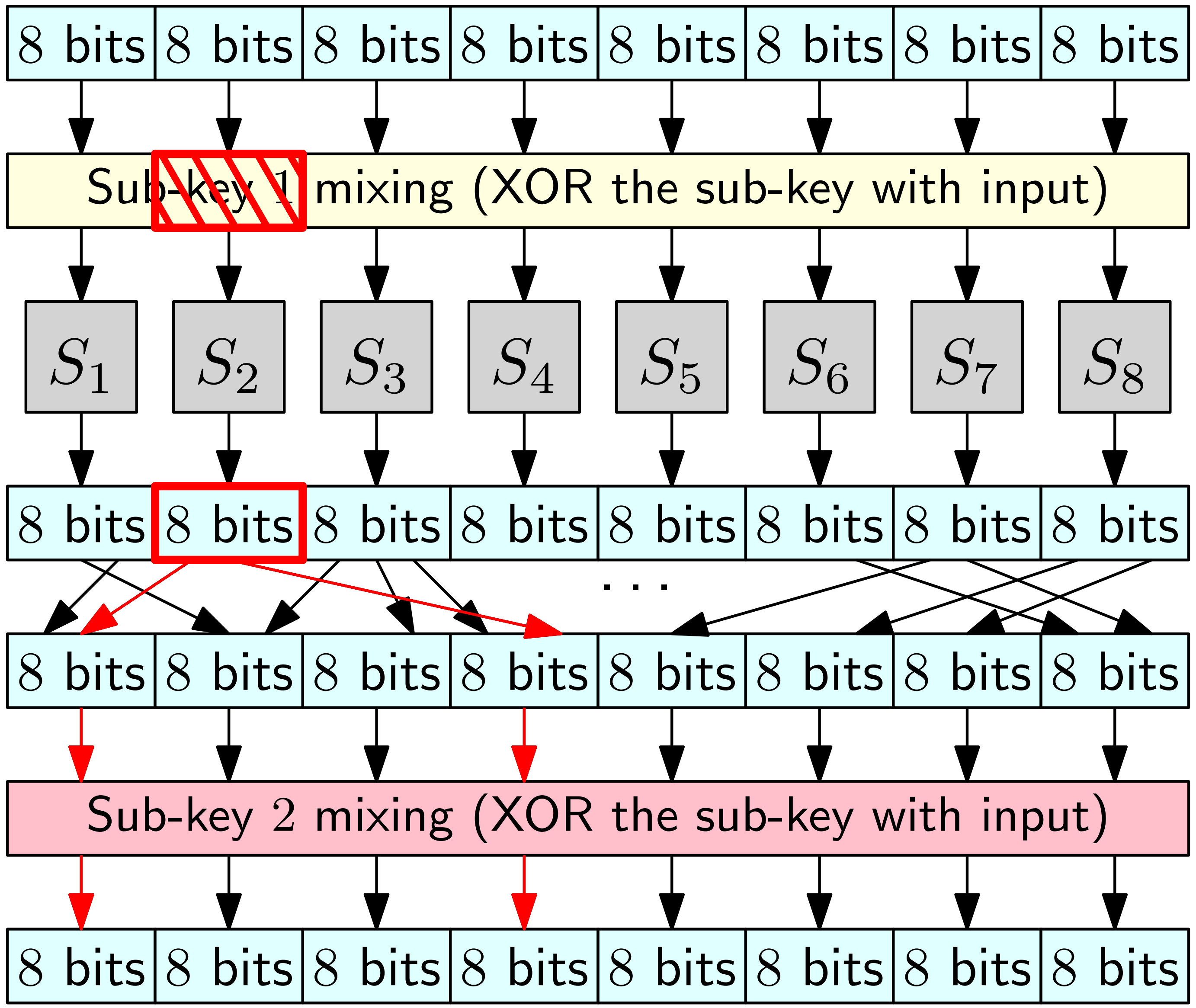
A better key recovery attack against a full 1-round SPN

- Guess only the part of 1st mixing sub-key that contributes to the input of some S-box
- This provides a candidate output value of the 1st S-box



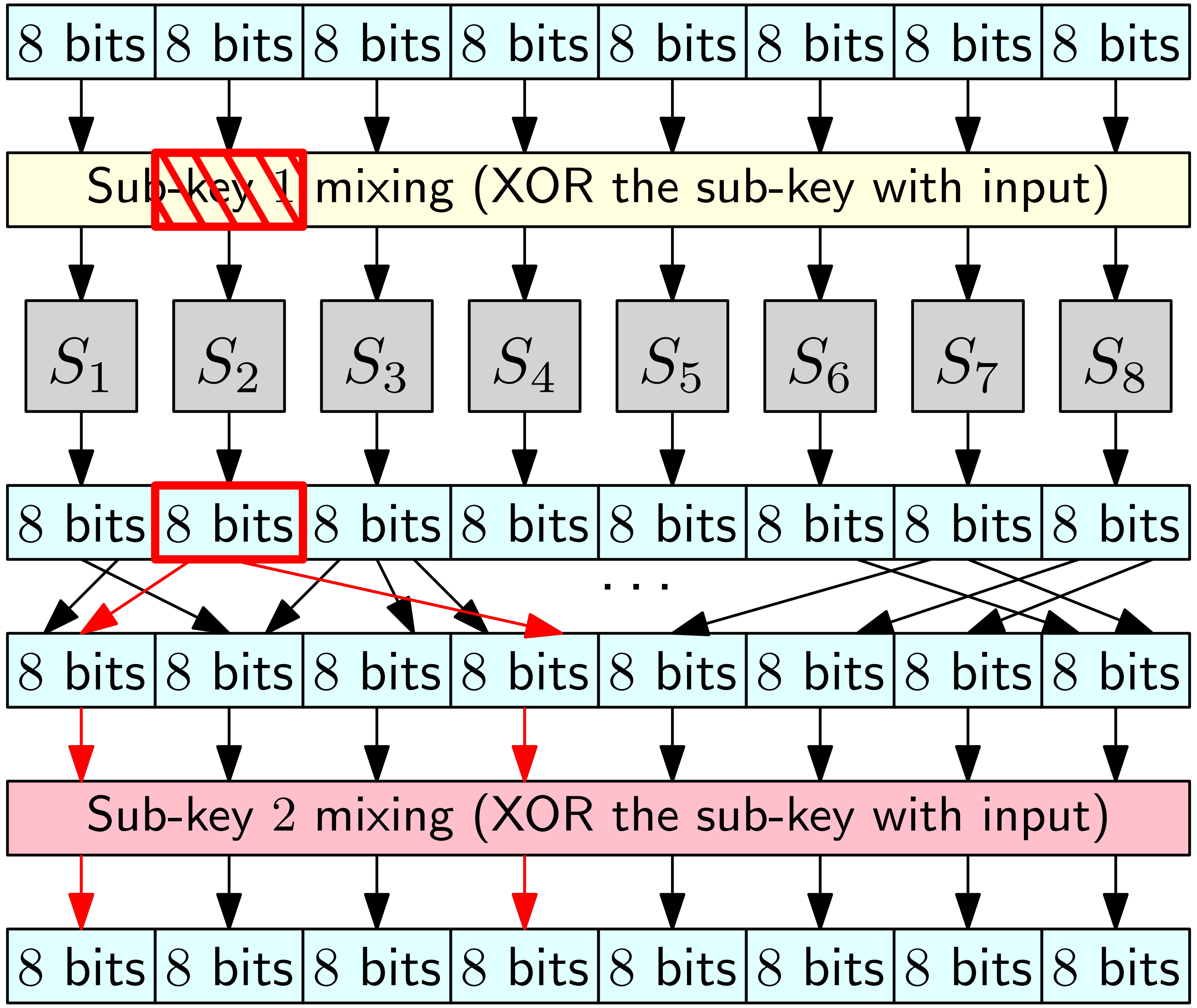
A better key recovery attack against a full 1-round SPN

- Guess only the part of 1st mixing sub-key that contributes to the input of some S-box
- This provides a candidate output value of the 1st S-box
- The output of the S-box is XOR-ed with some bits of the 2nd mixing sub-key to produce (part of) the output



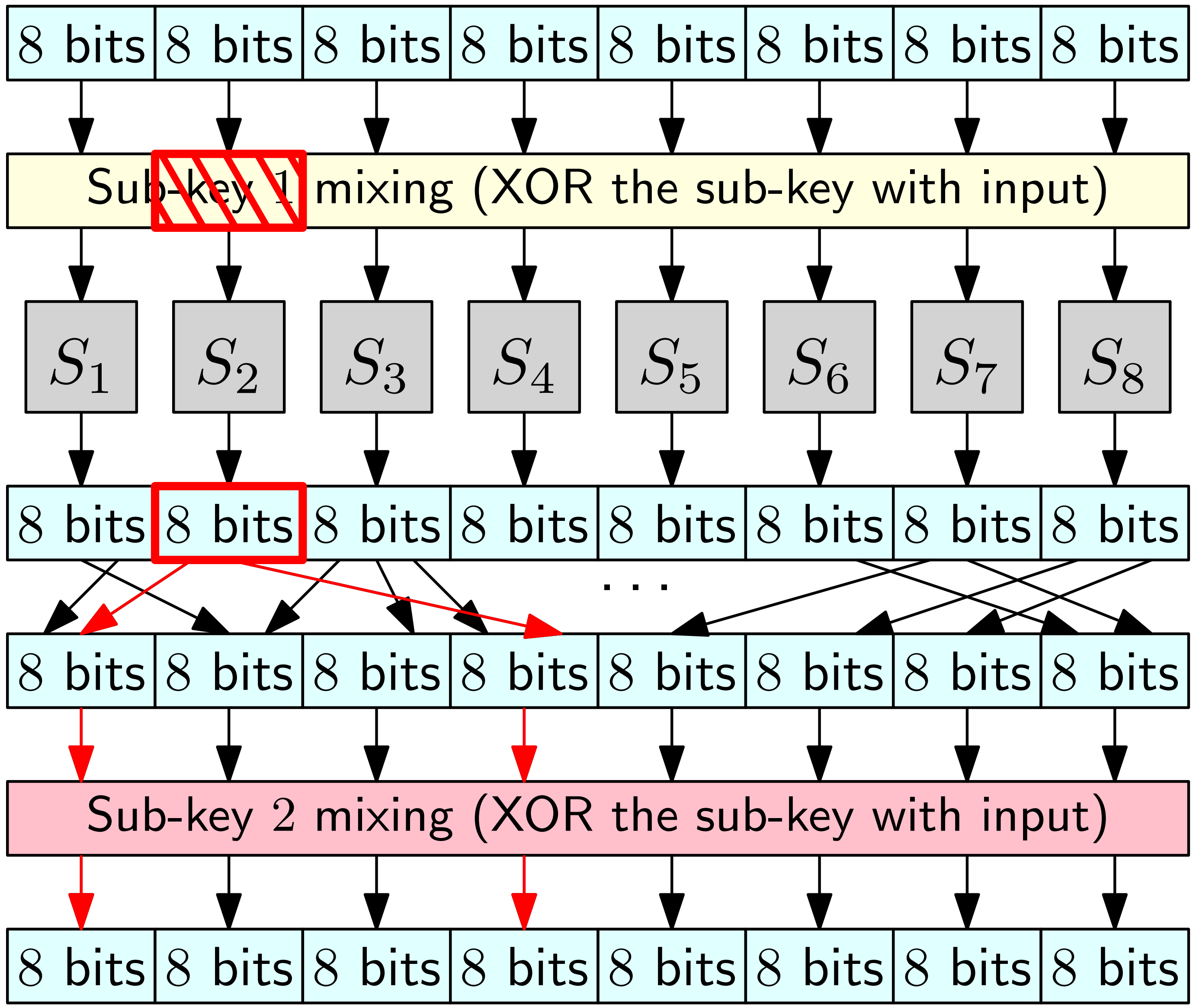
A better key recovery attack against a full 1-round SPN

- Guess only the part of 1st mixing sub-key that contributes to the input of some S-box
- This provides a candidate output value of the 1st S-box
- The output of the S-box is XOR-ed with some bits of the 2nd mixing sub-key to produce (part of) the output
- We know which bits of the 2nd mixing sub-key are used!



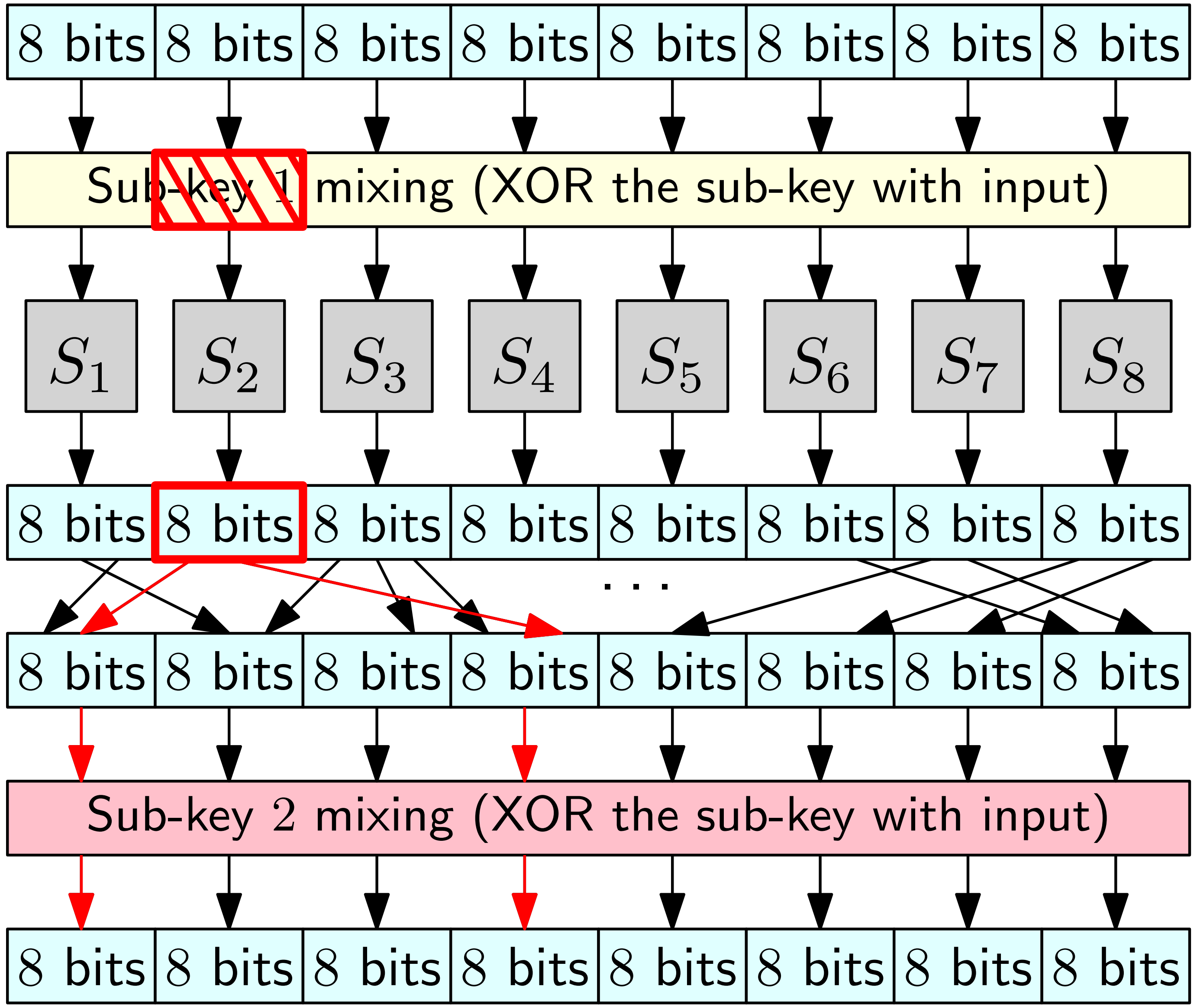
A better key recovery attack against a full 1-round SPN

- Guess only the part of 1st mixing sub-key that contributes to the input of some S-box
- This provides a candidate output value of the 1st S-box
- The output of the S-box is XOR-ed with some bits of the 2nd mixing sub-key to produce (part of) the output
- We know which bits of the 2nd mixing sub-key are used!
- We can recover the value of these bits by XOR-ing the S-box output with the corresponding bits of y



A better key recovery attack against a full 1-round SPN

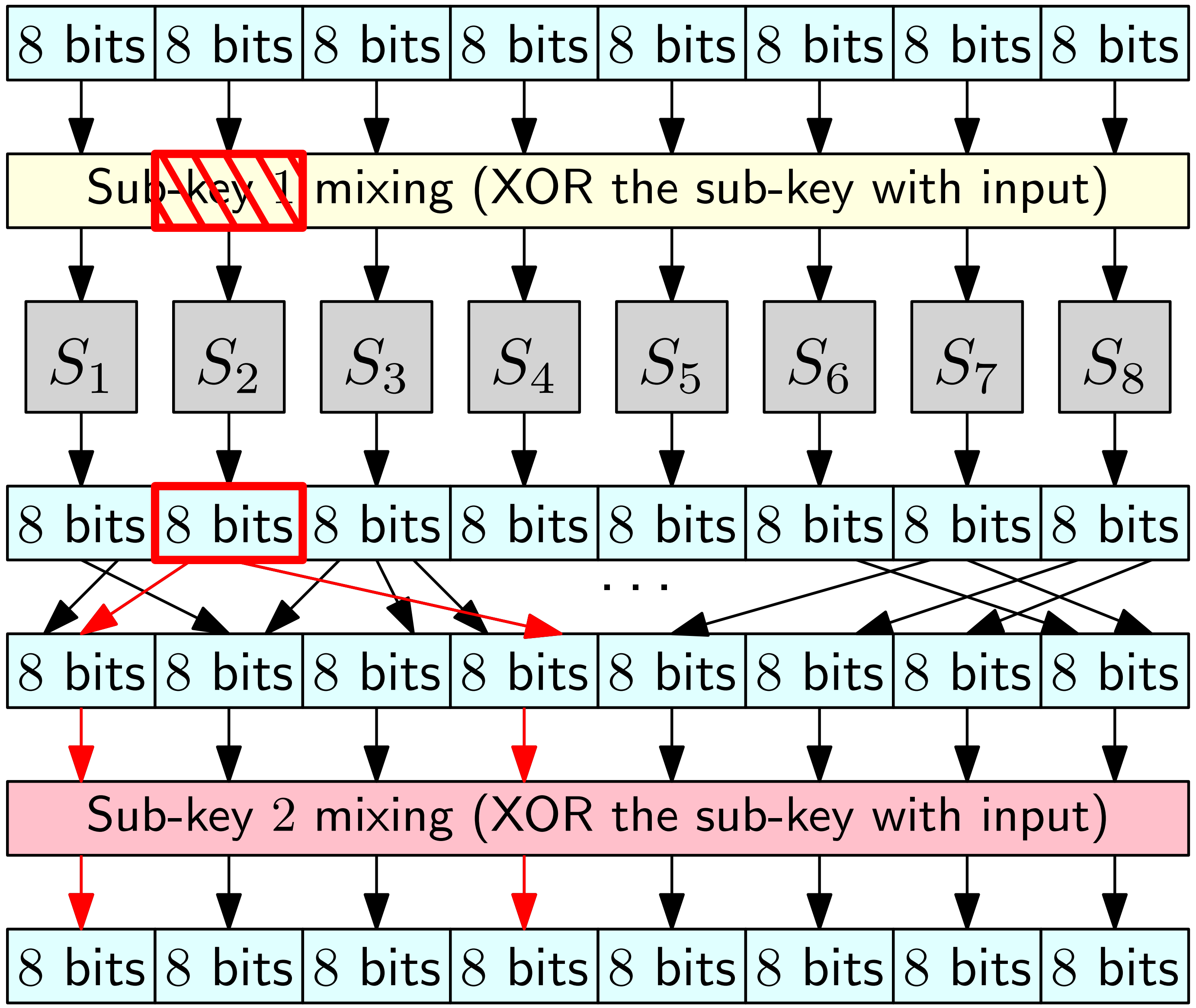
- Guess only the part of 1st mixing sub-key that contributes to the input of some S-box
- This provides a candidate output value of the 1st S-box
- The output of the S-box is XOR-ed with some bits of the 2nd mixing sub-key to produce (part of) the output
- We know which bits of the 2nd mixing sub-key are used!
- We can recover the value of these bits by XOR-ing the S-box output with the corresponding bits of y
- Each guess produces a candidate value for some bits in the 2nd mixing sub-key: use multiple input-output pairs to find the right one



A better key recovery attack against a full 1-round SPN

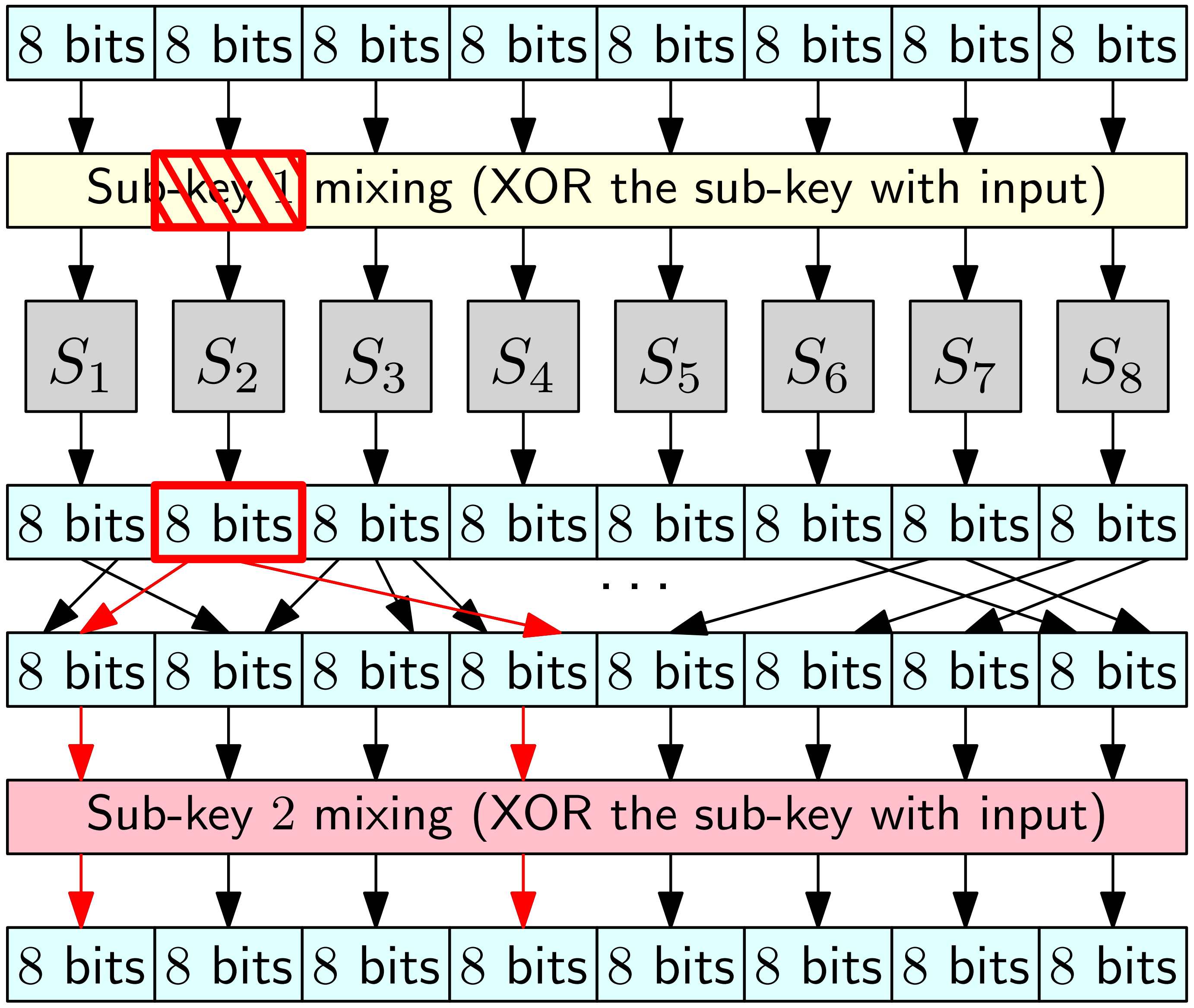
- Guess only the part of 1st mixing sub-key that contributes to the input of some S-box
- This provides a candidate output value of the 1st S-box
- The output of the S-box is XOR-ed with some bits of the 2nd mixing sub-key to produce (part of) the output
- We know which bits of the 2nd mixing sub-key are used!
- We can recover the value of these bits by XOR-ing the S-box output with the corresponding bits of y
- Each guess produces a candidate value for some bits in the 2nd mixing sub-key: use multiple input-output pairs to find the right one

We can break each group of key bits independently! (Repeat for each S-box)



A better key recovery attack against a full 1-round SPN

- Guess only the part of 1st mixing sub-key that contributes to the input of some S-box
- This provides a candidate output value of the 1st S-box
- The output of the S-box is XOR-ed with some bits of the 2nd mixing sub-key to produce (part of) the output
- We know which bits of the 2nd mixing sub-key are used!
- We can recover the value of these bits by XOR-ing the S-box output with the corresponding bits of y
- Each guess produces a candidate value for some bits in the 2nd mixing sub-key: use multiple input-output pairs to find the right one



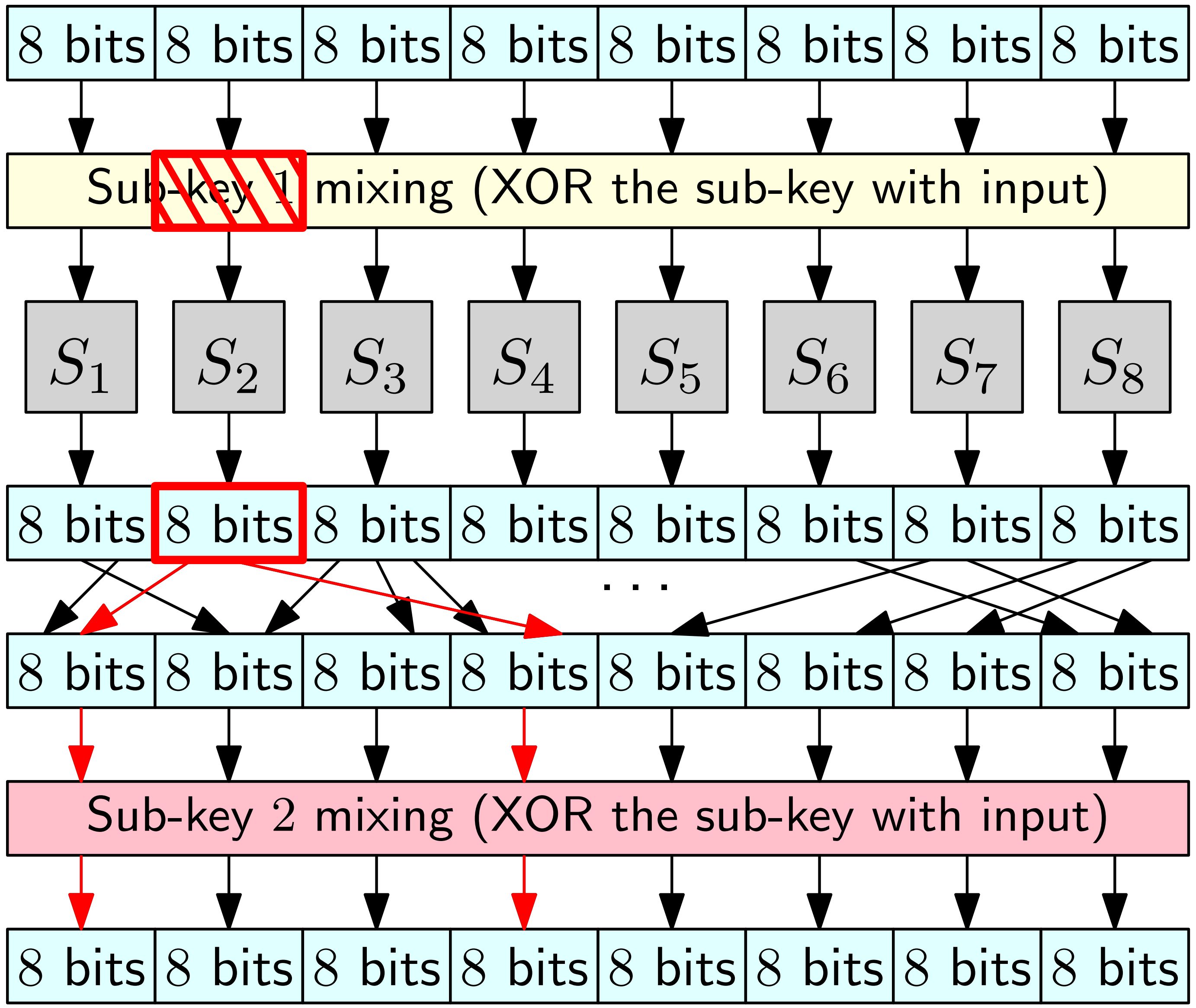
Time: $\approx$ #S-boxes $\cdot 2^{n/\#\text{S-boxes}}$

We can break each group of key bits independently!
(Repeat for each S-box)

A better key recovery attack against a full 1-round SPN

- Guess only the part of 1st mixing sub-key that contributes to the input of some S-box
- This provides a candidate output value of the 1st S-box
- The output of the S-box is XOR-ed with some bits of the 2nd mixing sub-key to produce (part of) the output
- We know which bits of the 2nd mixing sub-key are used!
- We can recover the value of these bits by XOR-ing the S-box output with the corresponding bits of y
- Each guess produces a candidate value for some bits in the 2nd mixing sub-key: use multiple input-output pairs to find the right one

We can break each group of key bits independently!
(Repeat for each S-box)



Time: $\approx$ #S-boxes $\cdot 2^{n/\text{\#S-boxes}}$

In the example: $\approx 8 \cdot 2^8 = 2^{11}$

(intead of 2^{64} of the previous attack or 2^{128} of a naive bruteforce)

Attacking more rounds

These attacks become more difficult as the number of rounds increases

Attacking more rounds

These attacks become more difficult as the number of rounds increases

The more the rounds, the more a small change in the input affects the whole output (avalanche effect)

Attacking more rounds

These attacks become more difficult as the number of rounds increases

The more the rounds, the more a small change in the input affects the whole output (avalanche effect)

At some points these attacks become impractical

Attacking more rounds

These attacks become more difficult as the number of rounds increases

The more the rounds, the more a small change in the input affects the whole output (avalanche effect)

At some points these attacks become impractical

Good block ciphers based on SPNs need to use a large enough number of rounds

Attacking more rounds

These attacks become more difficult as the number of rounds increases

The more the rounds, the more a small change in the input affects the whole output (avalanche effect)

At some points these attacks become impractical

Good block ciphers based on SPNs need to use a large enough number of rounds

This is just a necessary condition for security: If the S-boxes or the mixing permutation are poorly designed, the block cipher might still be insecure (regardless of the number of rounds)!

Attacking more rounds

These attacks become more difficult as the number of rounds increases

The more the rounds, the more a small change in the input affects the whole output (avalanche effect)

At some points these attacks become impractical

Good block ciphers based on SPNs need to use a large enough number of rounds

This is just a necessary condition for security: If the S-boxes or the mixing permutation are poorly designed, the block cipher might still be insecure (regardless of the number of rounds)!

It's common to see results of the form:

"A reduced version of [block cipher] using X instead of Y rounds has been broken"

Designing Block Ciphers

- To design a block cipher, we want the computed function to be "indistinguishable" from a uniform permutation over $\{0,1\}^\ell$

- If x and x' differ, even just by one bit, the outputs of $F_k(x)$ and $F_k(x')$ should look unrelated (except for $F_k(x) \neq F_k(x)$

- On average $\approx \ell/2$ bits change between $F_k(x)$ and $F_k(x')$

- The position of the changing bits looks "random"

How do we achieve this?

- Substitution Permutation Networks (SPNs)

- Feistel Networks

Designing Block Ciphers

- To design a block cipher, we want the computed function to be “indistinguishable” from a uniform permutation over $\{0,1\}^\ell$
- If x and x' differ, even just by one bit, the outputs of $F_k(x)$ and $F_k(x')$ should look unrelated (except for $F_k(x) \neq F_k(x)$)
- On average $\approx \ell/2$ bits change between $F_k(x)$ and $F_k(x')$
- The position of the changing bits looks “random”

How do we achieve this?

- Substitution Permutation Networks (SPNs)
- Feistel Networks

(Balanced) Feistel Networks

- Alternative approach to SPNs to build block ciphers
- Use non-invertible components to build an invertible permutation

(Balanced) Feistel Networks

- Alternative approach to SPNs to build block ciphers
- Use non-invertible components to build an invertible permutation
- Just like SPNs, Fesistel networks work in multiple rounds
- Each round uses a keyed round function

(Balanced) Feistel Networks

- Alternative approach to SPNs to build block ciphers
- Use non-invertible components to build an invertible permutation
- Just like SPNs, Fesistel networks work in multiple rounds
- Each round uses a keyed round function

Not necessarily invertible!

(Balanced) Feistel Networks

- Alternative approach to SPNs to build block ciphers

- Use non-invertible components to build an invertible permutation

- Just like SPNs, Fesistel networks work in multiple rounds

- Each round uses a keyed round function

Not necessarily invertible!

- The keys of the round functions are the **sub-keys** determined by a **master key** of the whole block cipher

(Balanced) Feistel Networks

- Alternative approach to SPNs to build block ciphers
- Use non-invertible components to build an invertible permutation
- Just like SPNs, Fesistel networks work in multiple rounds
- Each round uses a keyed round function

Not necessarily invertible!

- The keys of the round functions are the **sub-keys** determined by a **master key** of the whole block cipher
- Let ℓ be the block length. The keyed round function for the i-th round is

$$\widehat{f_i} : \{0,1\}^n \times \{0,1\}^{\ell/2} \to \{0,1\}^{\ell/2}$$

(Balanced) Feistel Networks

- Alternative approach to SPNs to build block ciphers
- Use non-invertible components to build an invertible permutation
- Just like SPNs, Fesistel networks work in multiple rounds
- Each round uses a keyed round function

Not necessarily invertible!

- The keys of the round functions are the **sub-keys** determined by a **master key** of the whole block cipher
- Let ℓ be the block length. The keyed round function for the i-th round is

$$\widehat{f_i} : \{0,1\}^n \times \{0,1\}^{\ell/2} \to \{0,1\}^{\ell/2}$$

(Balanced) Feistel Networks

- Alternative approach to SPNs to build block ciphers

- Use non-invertible components to build an invertible permutation

- Just like SPNs, Fesistel networks work in multiple rounds

- Each round uses a keyed round function

Not necessarily invertible!

- The keys of the round functions are the **sub-keys** determined by a **master key** of the whole block cipher

- Let ℓ be the block length. The keyed round function for the i-th round is

$$\widehat{f}_i : \{0,1\}^n \times \{0,1\}^{\ell/2} \to \{0,1\}^{\ell/2}$$

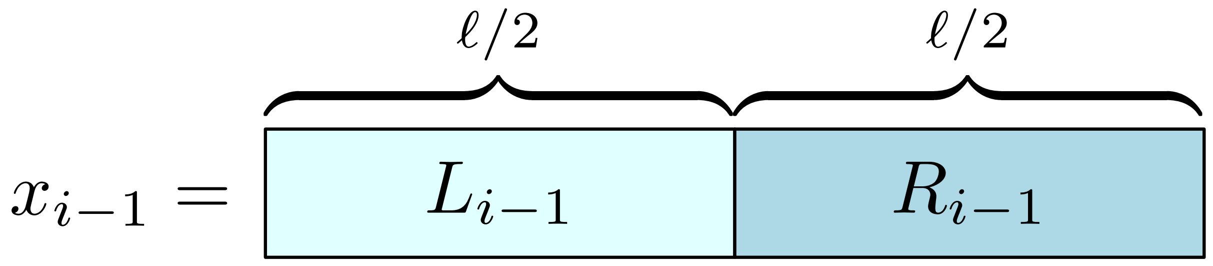
- To keep notation simple, define $f_i : \{0,1\}^{\ell/2} \to \{0,1\}^{\ell/2}$ as $f_i(x) = \widehat{f}(k_i, x)$, where k_i is the i-th sub-key

Feistel Network Rounds

- Let x_{i-1} and x_i be the input and the output of the i-th round of the Feistel Network, respectively

Feistel Network Rounds

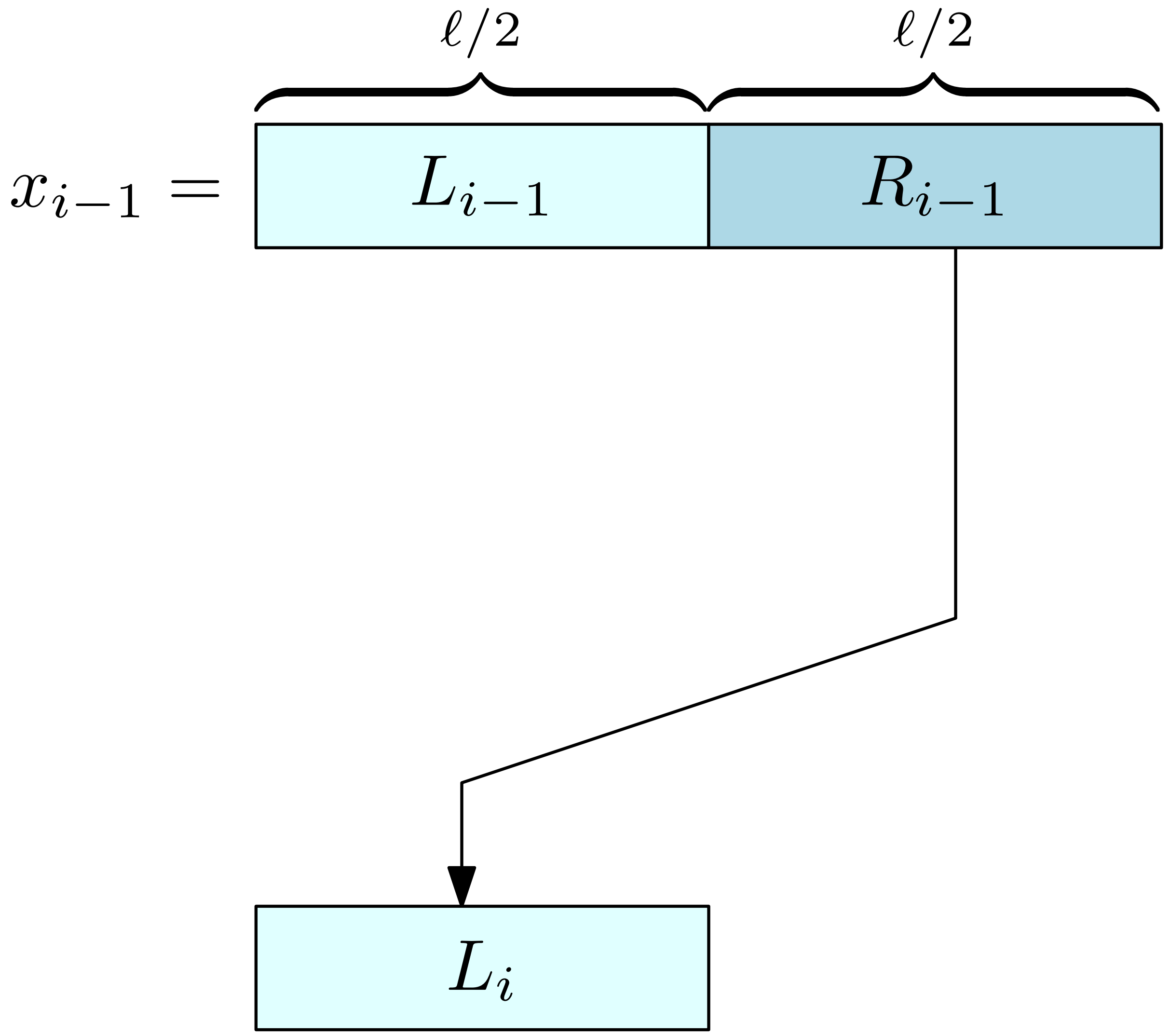
- Let x_{i-1} and x_i be the input and the output of the i-th round of the Feistel Network, respectively
- Split each x_i into a "left side" L_i and a right side R_i, each of length $\ell/2$
  - $x_{i-1} = L_{i-1} \,\|\, R_{i-1}$



Feistel Network Rounds

- Let x_{i-1} and x_i be the input and the output of the i-th round of the Feistel Network, respectively
- Split each x_i into a “left side” L_i and a right side R_i, each of length $\ell/2$

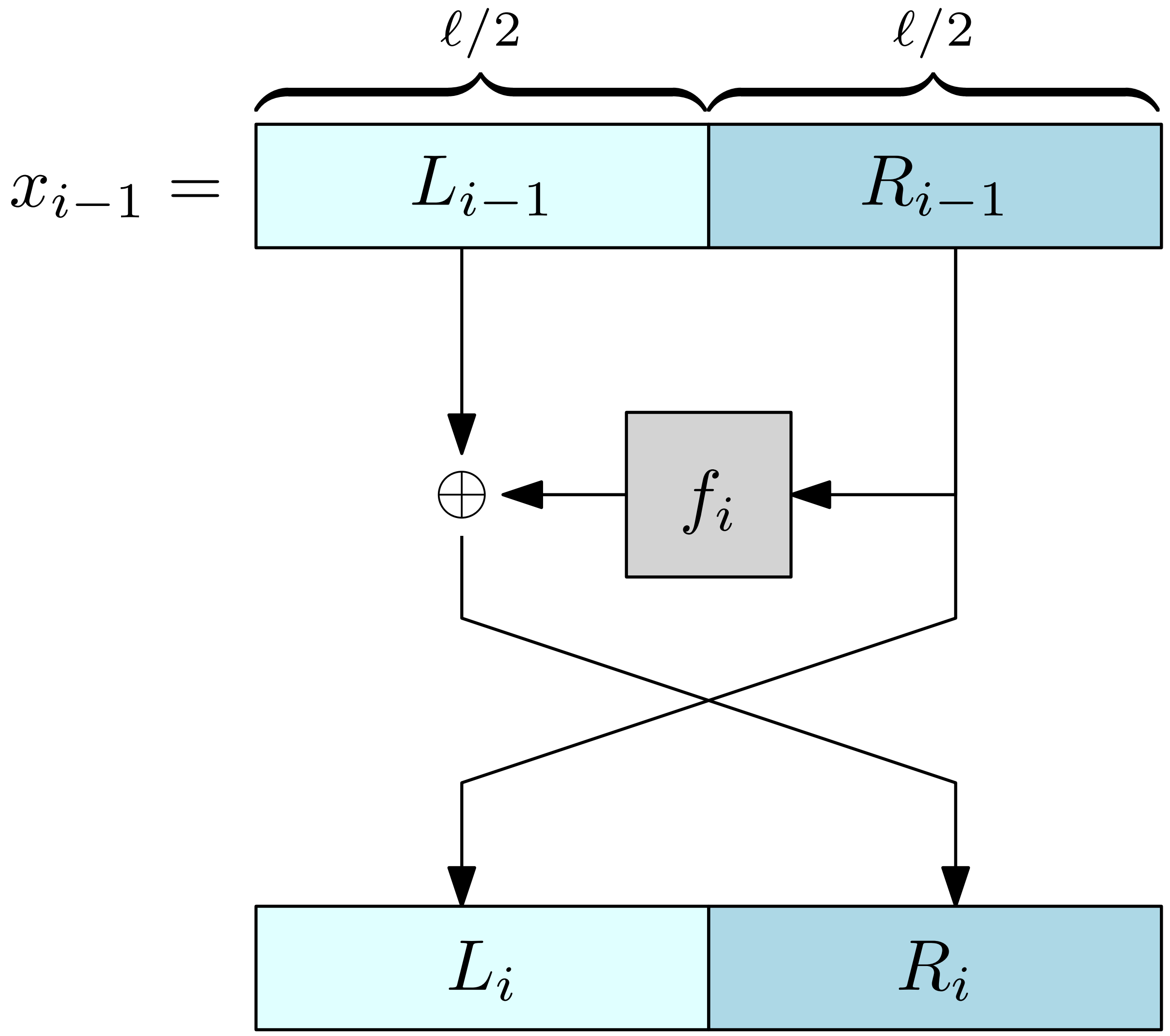
  - $x_{i-1} = L_{i-1} \,\|\, R_{i-1}$
  - $L_i = R_{i-1}$



Feistel Network Rounds

- Let x_{i-1} and x_i be the input and the output of the i-th round of the Feistel Network, respectively
- Split each x_i into a "left side" L_i and a right side R_i, each of length $\ell/2$

  - $x_{i-1} = L_{i-1} \parallel R_{i-1}$
  - $L_i = R_{i-1}$
  - $R_i = L_{i-1} \oplus f_i(R_{i-1})$

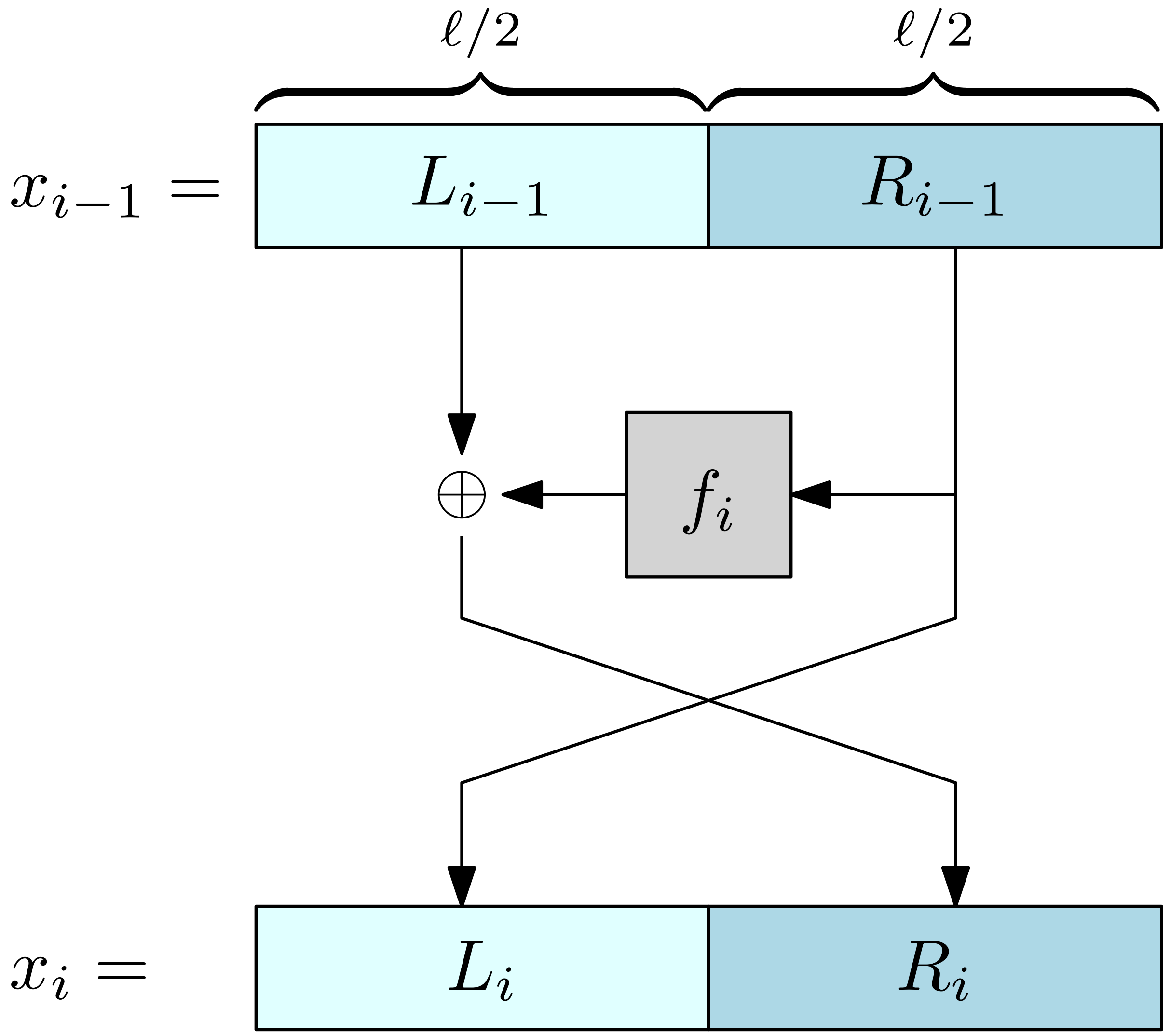


Feistel Network Rounds

- Let x_{i-1} and x_i be the input and the output of the i-th round of the Feistel Network, respectively

- Split each x_i into a "left side" L_i and a right side R_i, each of length $\ell/2$

  - $x_{i-1} = L_{i-1} \,\|\, R_{i-1}$
  - $L_i = R_{i-1}$
  - $R_i = L_{i-1} \oplus f_i(R_{i-1})$
  - $x_i = L_i \,\|\, R_i$

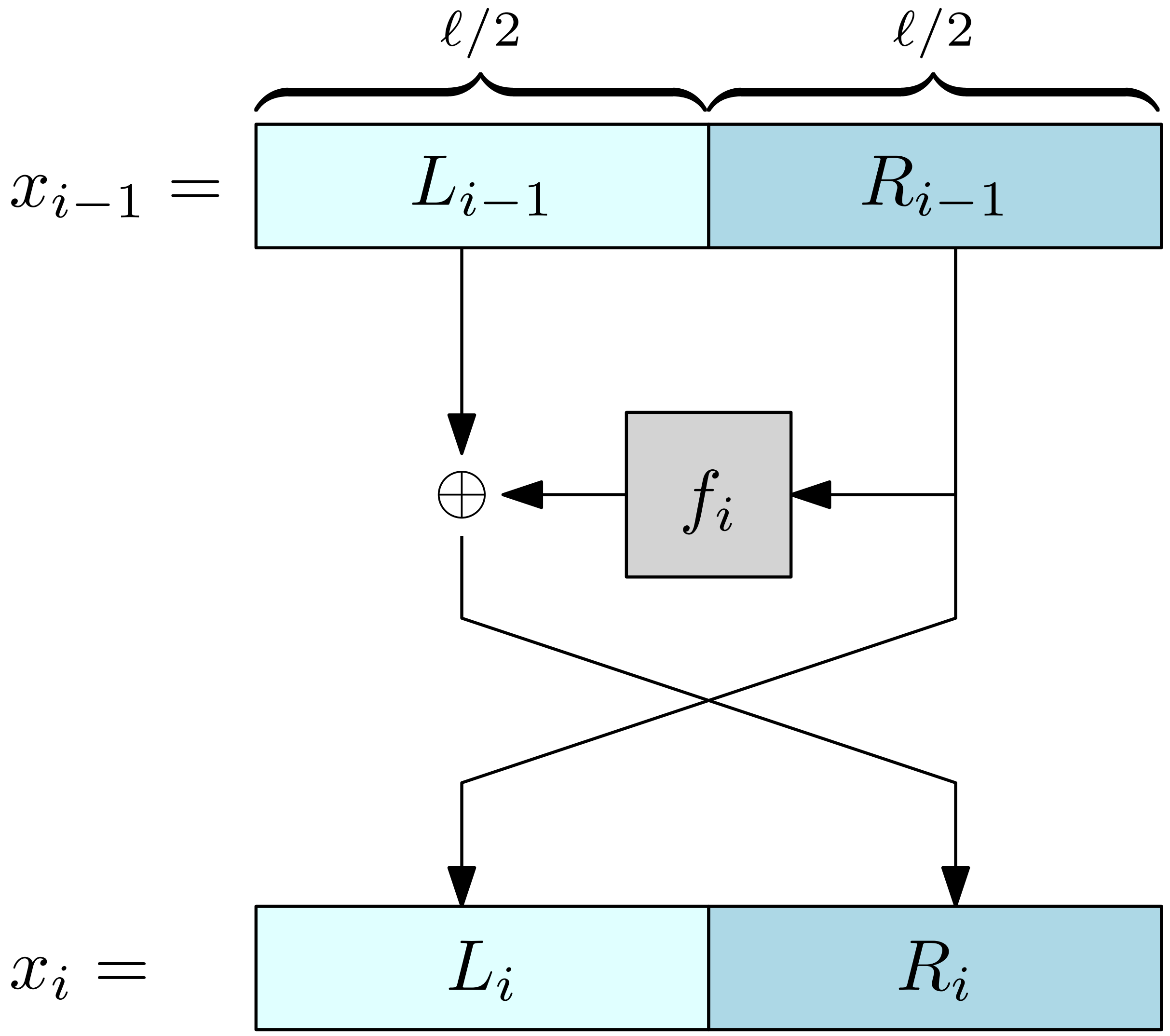


Feistel Network Rounds

- Let x_{i-1} and x_i be the input and the output of the i-th round of the Feistel Network, respectively

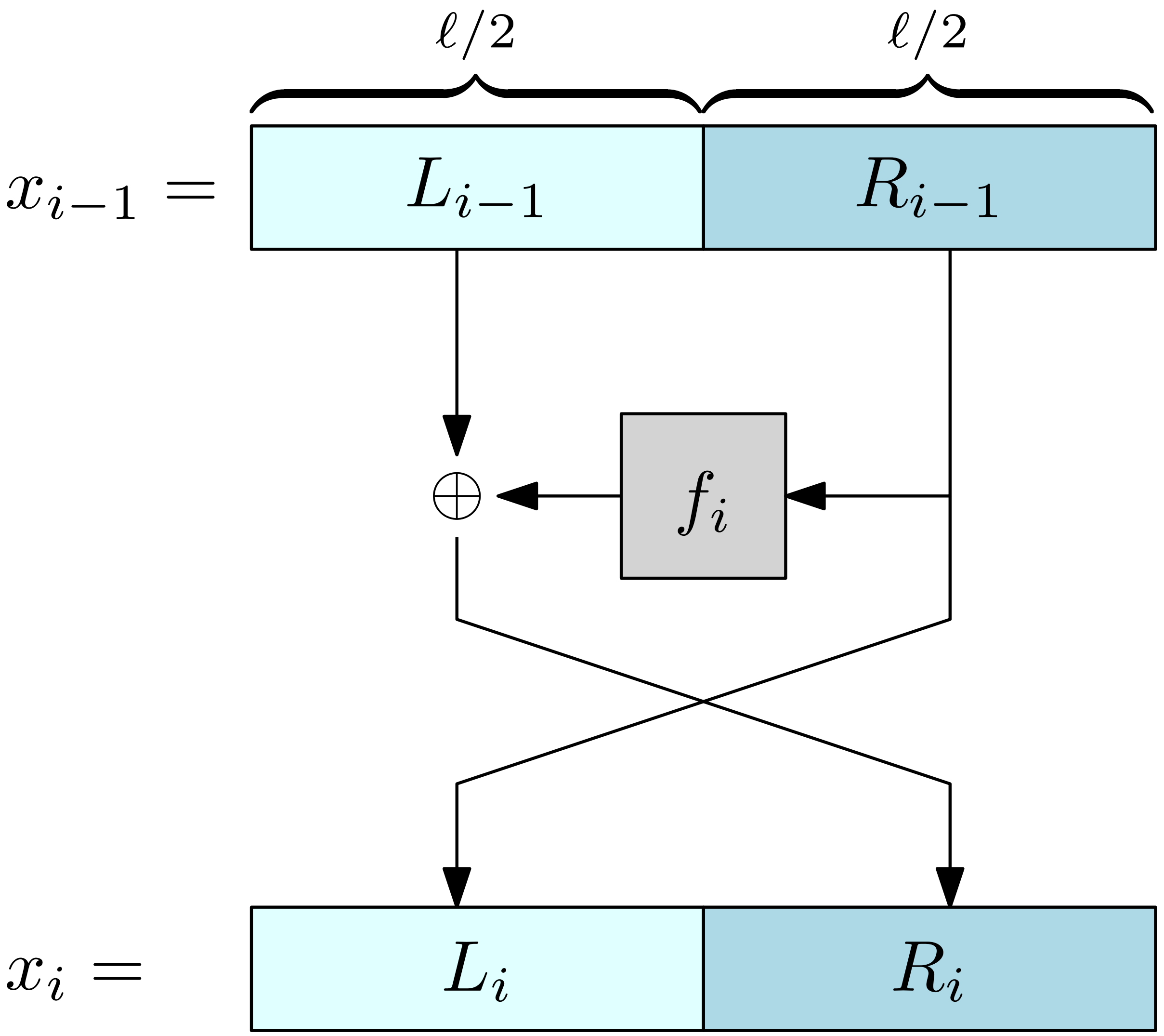
- Split each x_i into a "left side" L_i and a right side R_i, each of length $\ell/2$

- $x_{i-1} = L_{i-1} \| R_{i-1}$
- $L_i = R_{i-1}$
- $R_i = L_{i-1} \oplus f_i(R_{i-1})$
- $x_i = L_i \| R_i$



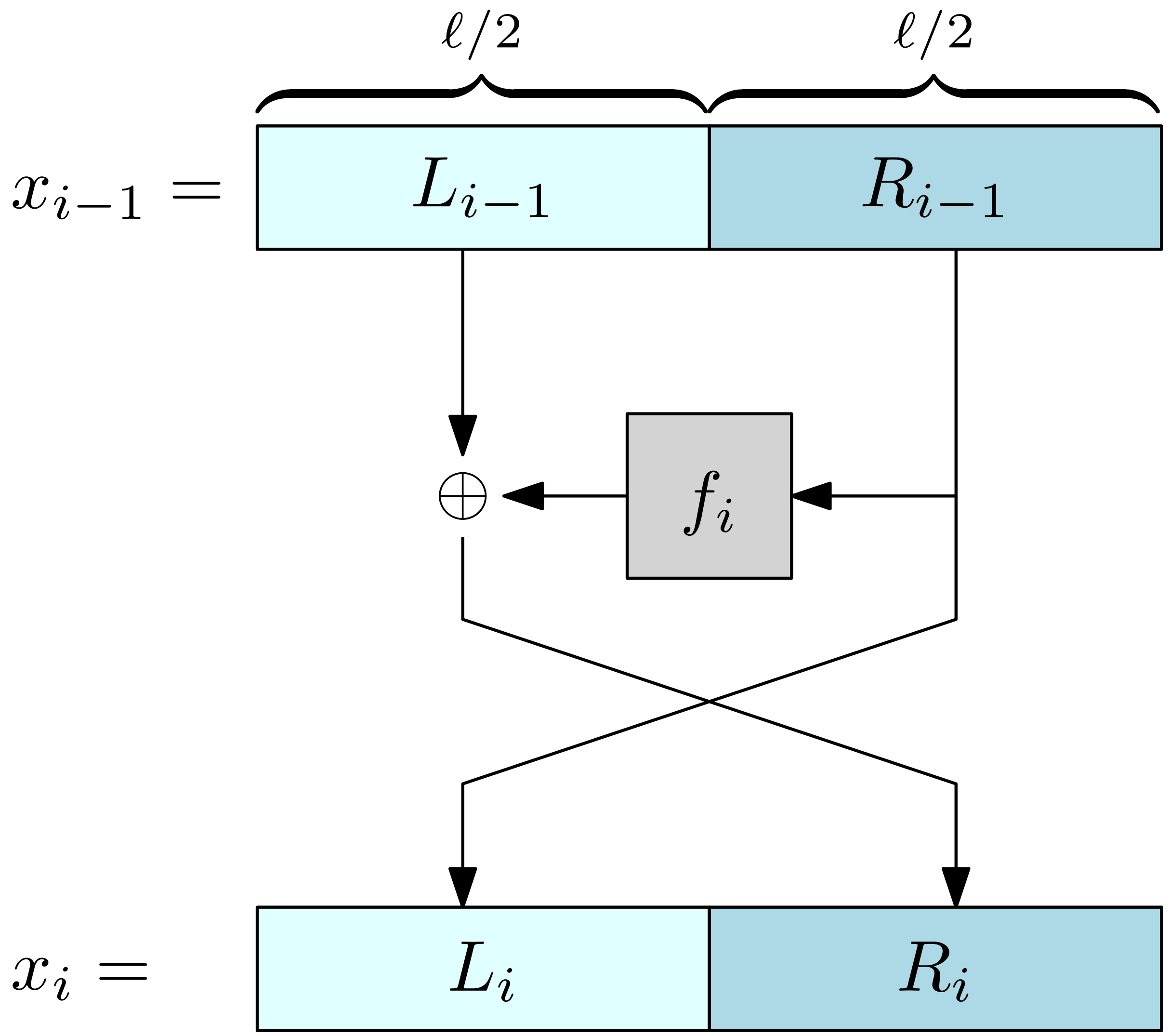
Is a Feistel Network round invertible? (How?)

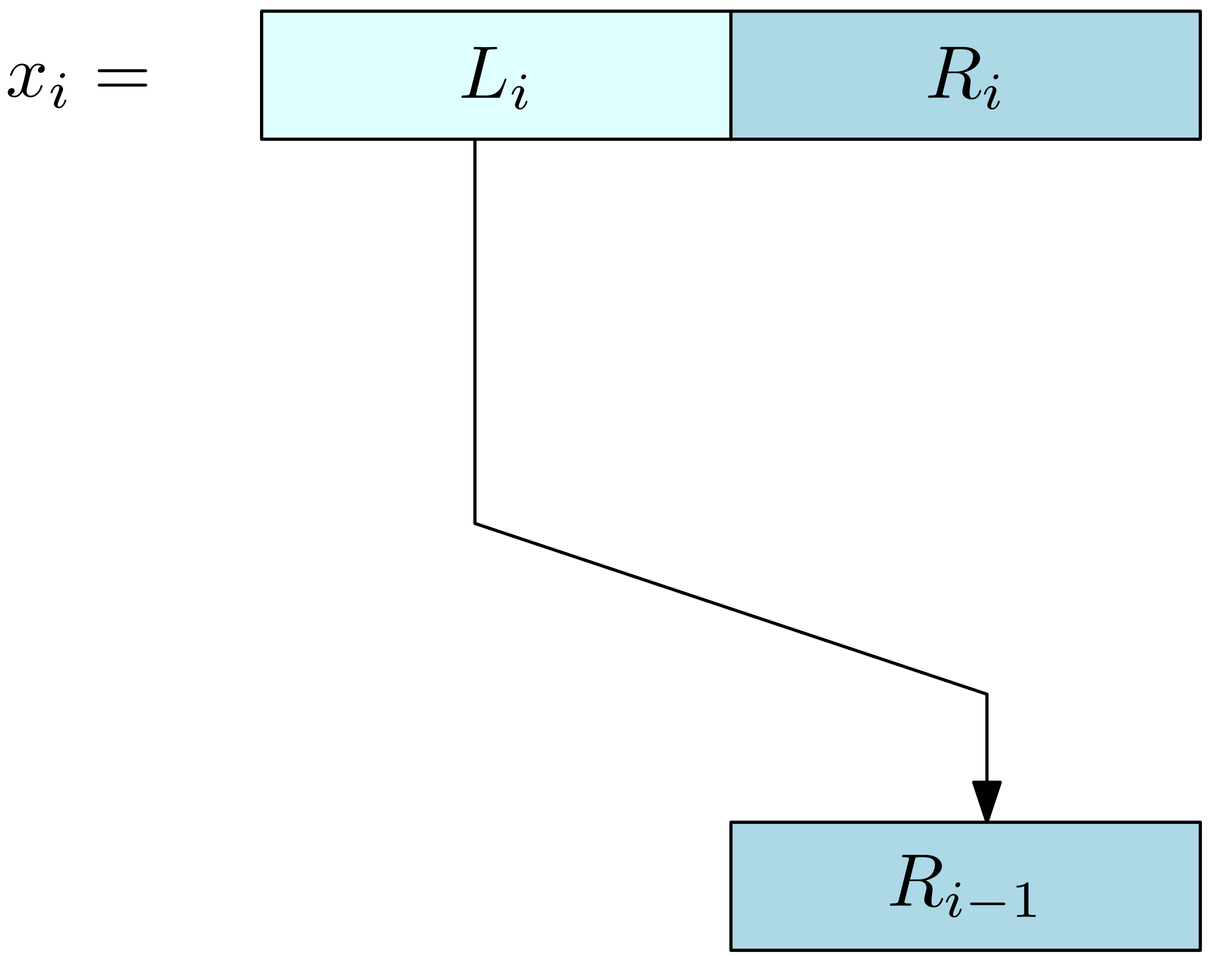
Inverting a Round of Feistel Network



- $L_i = R_{i-1}$
- $R_i = L_{i-1} \oplus f_i(R_{i-1})$

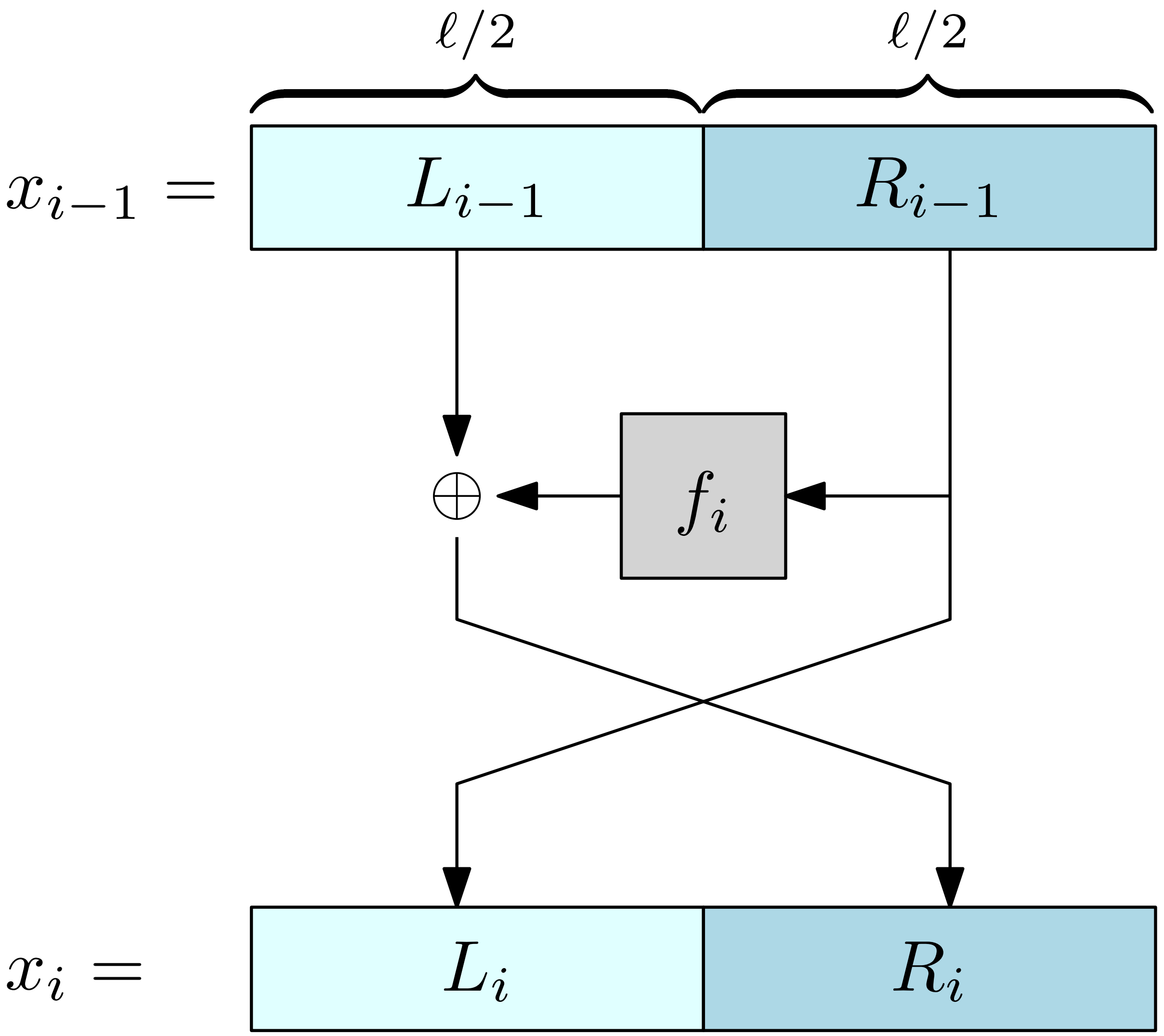
Inverting a Round of Feistel Network

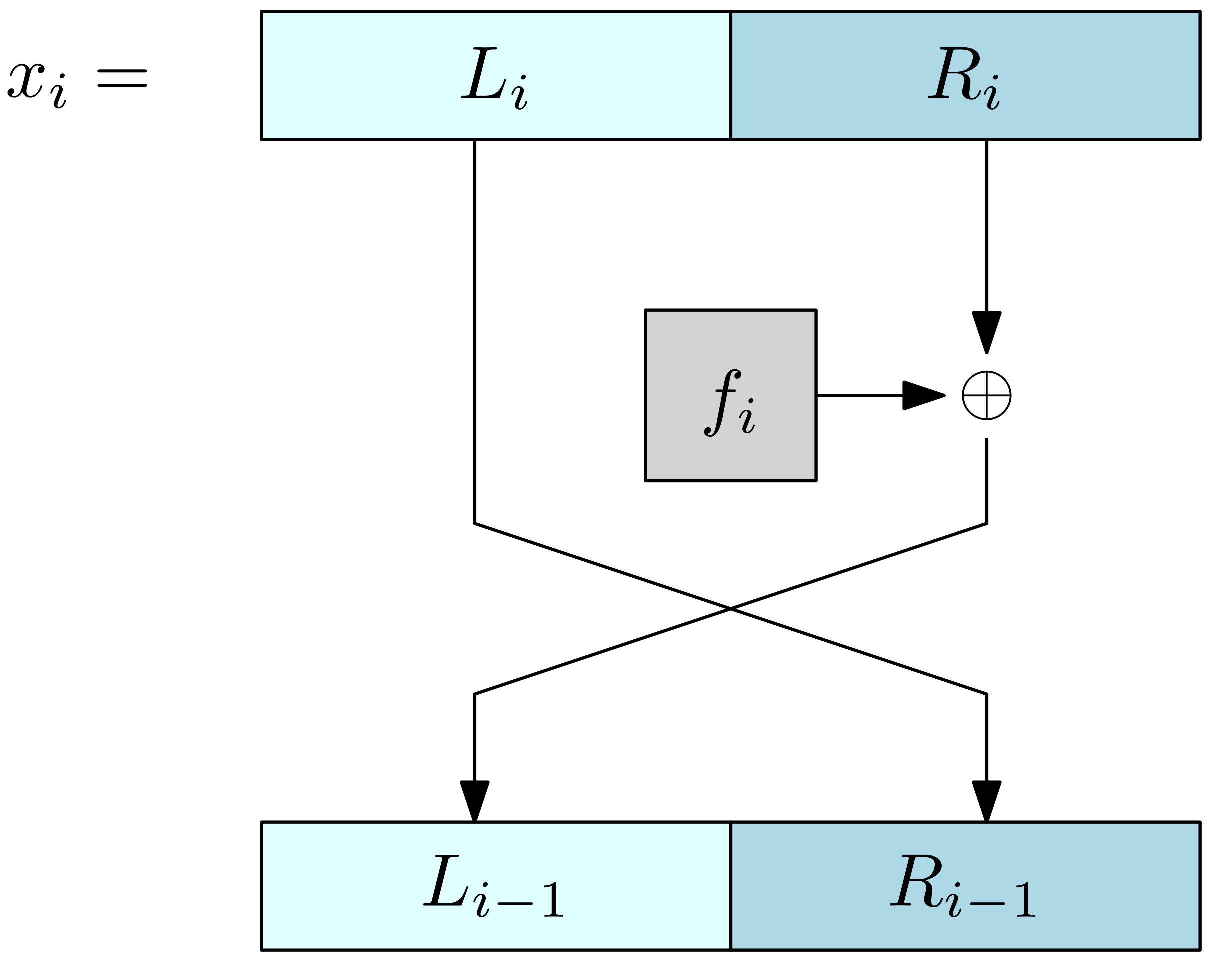




- $L_i = R_{i-1}$
- $R_i = L_{i-1} \oplus f_i(R_{i-1})$
- $R_{i-1} = L_i$

Inverting a Round of Feistel Network

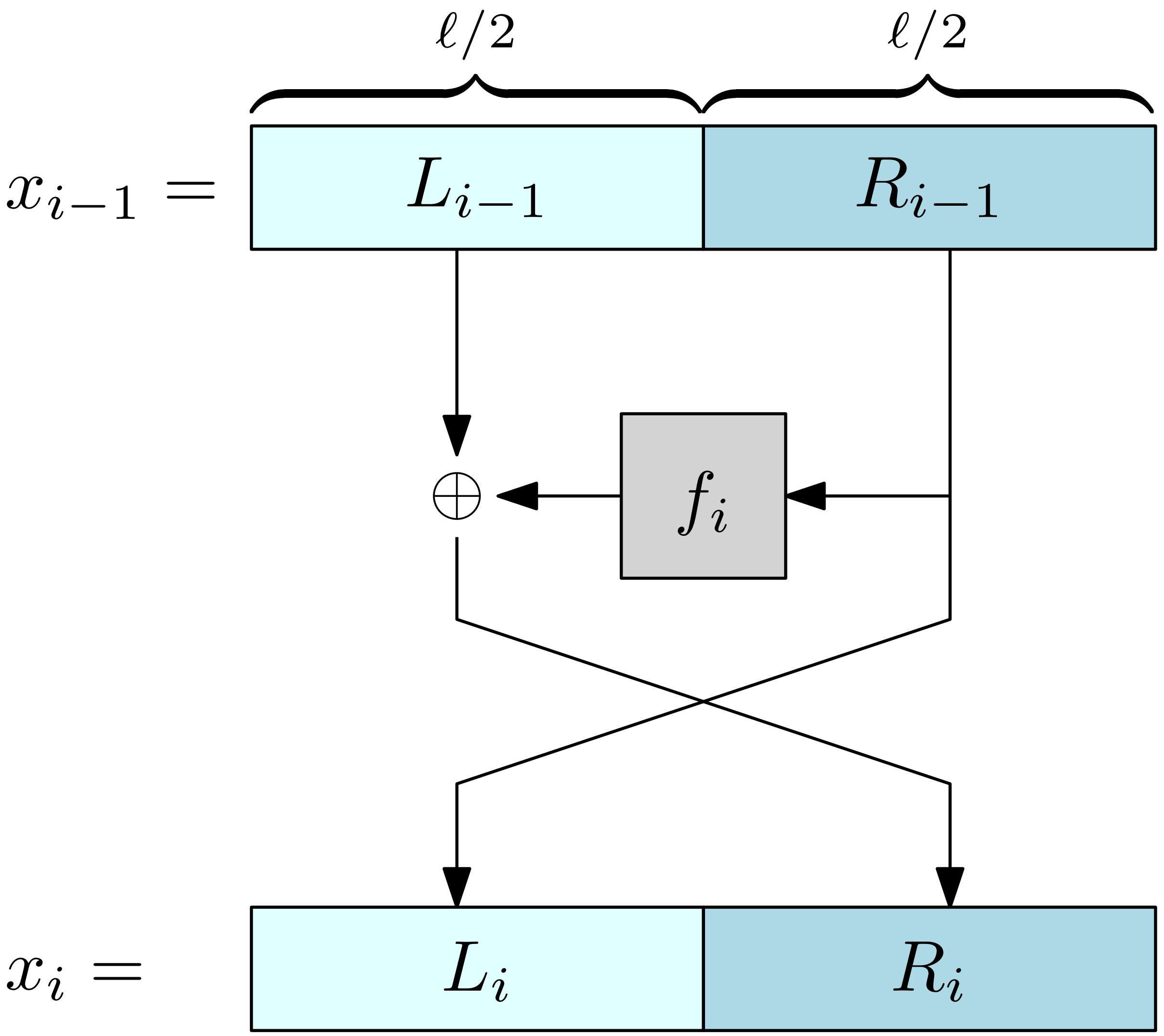


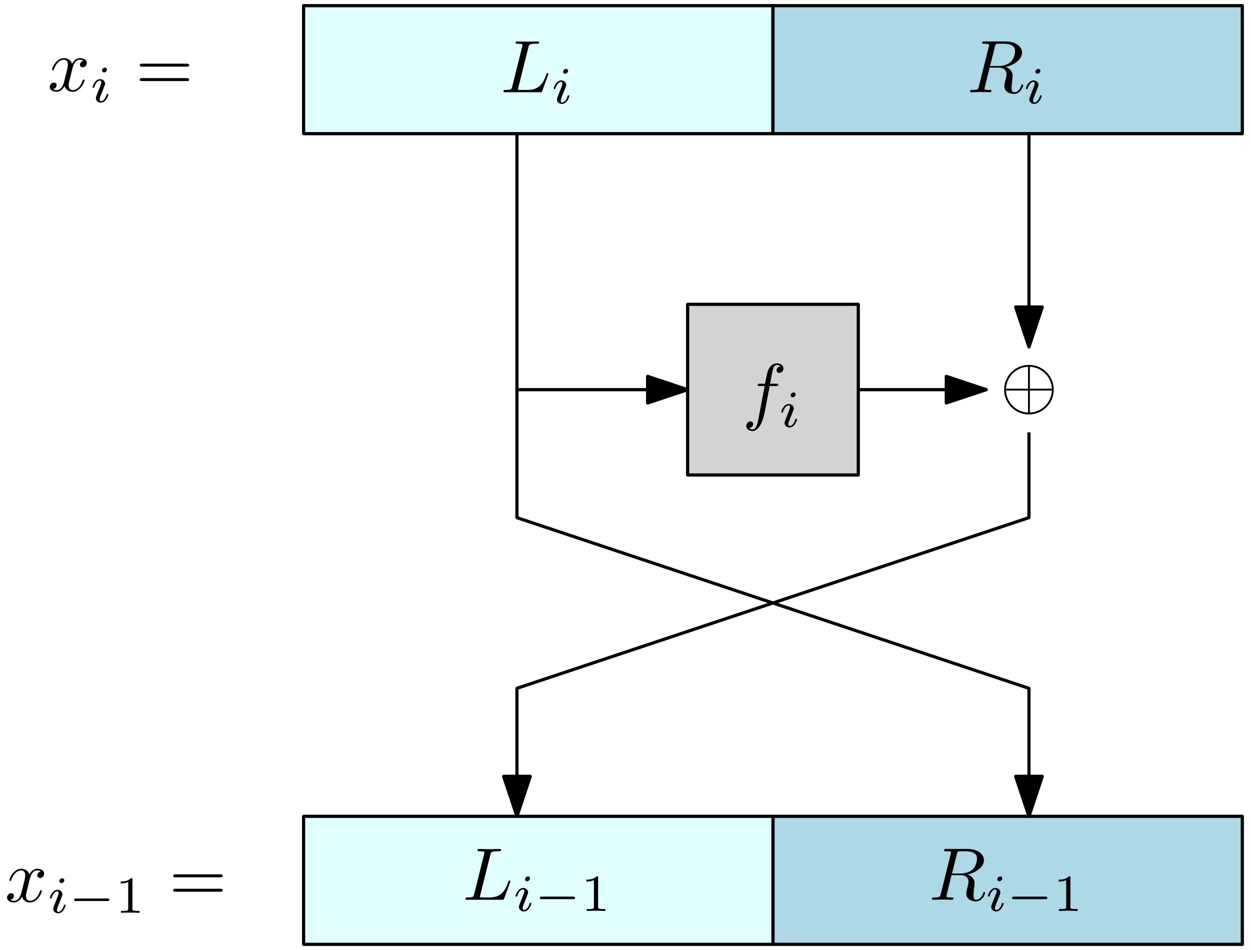


- $L_i = R_{i-1}$
- $R_i = L_{i-1} \oplus f_i(R_{i-1})$

- $R_{i-1} = L_i$
- $L_{i-1} = R_i \oplus f_i(R_{i-1})$

Inverting a Round of Feistel Network

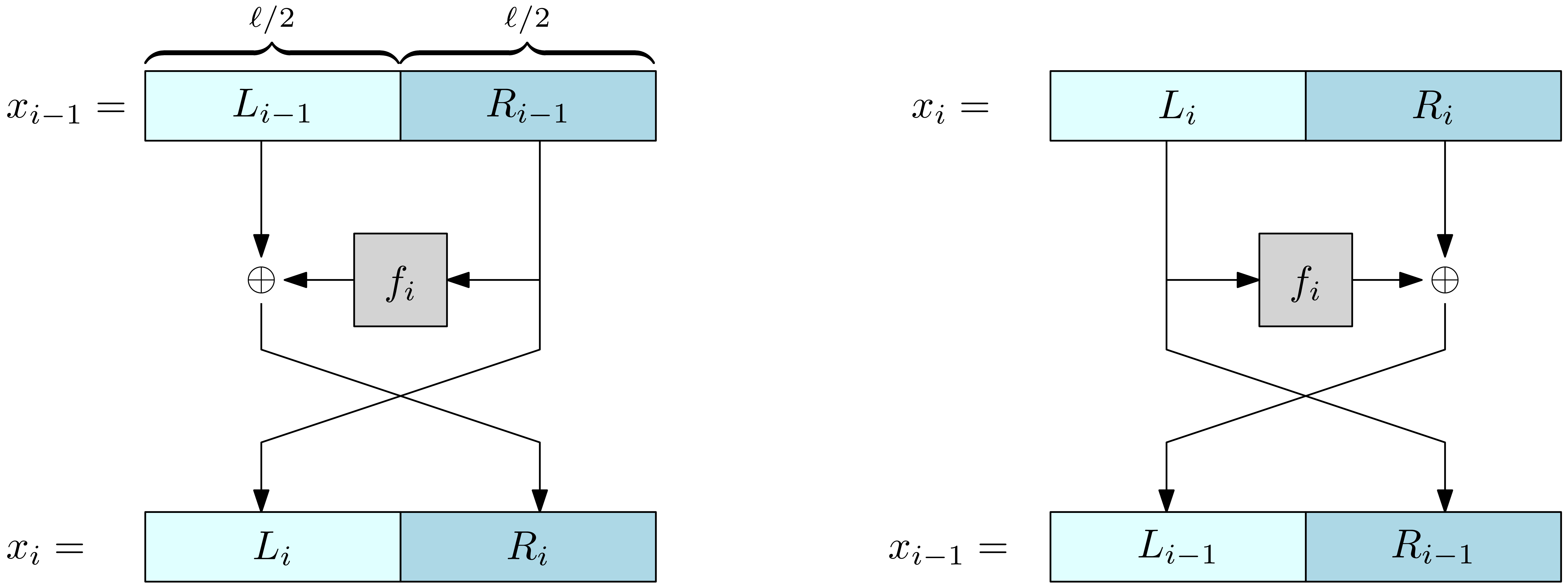




- $L_i = R_{i-1}$
- $R_i = L_{i-1} \oplus f_i(R_{i-1})$

- $R_{i-1} = L_i$
- $L_{i-1} = R_i \oplus f_i(R_{i-1}) \;= R_i \oplus f_i(L_i)$

Inverting a Round of Feistel Network

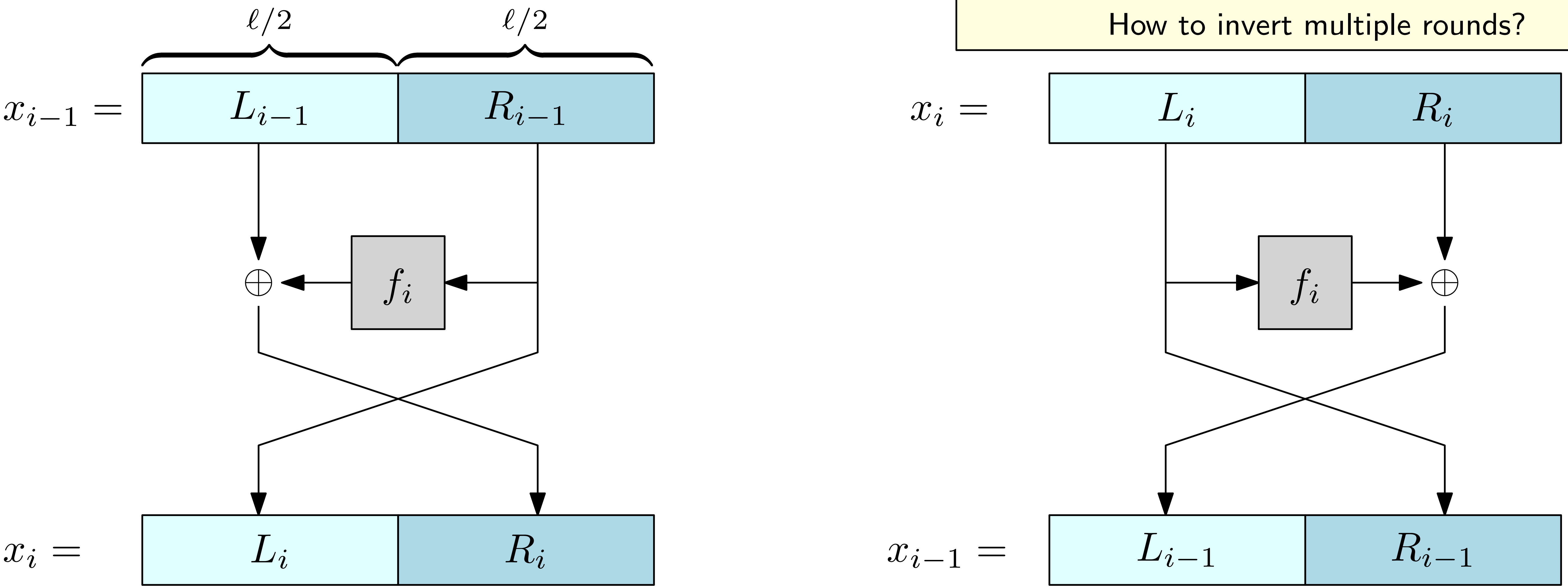


Let F be a keyed function defined by a Feistel network. Then regardless of the key schedule, the round functions $\widehat{f_i}$, and the number of rounds, F_k is a permutation for any k.

Inverting a Round of Feistel Network

F^{-1} is the same as F once the "left" and "right" sides are swapped!

How to invert multiple rounds?



Let F be a keyed function defined by a Feistel network. Then regardless of the key schedule, the round functions $\widehat{f_i}$, and the number of rounds, F_k is a permutation for any k.

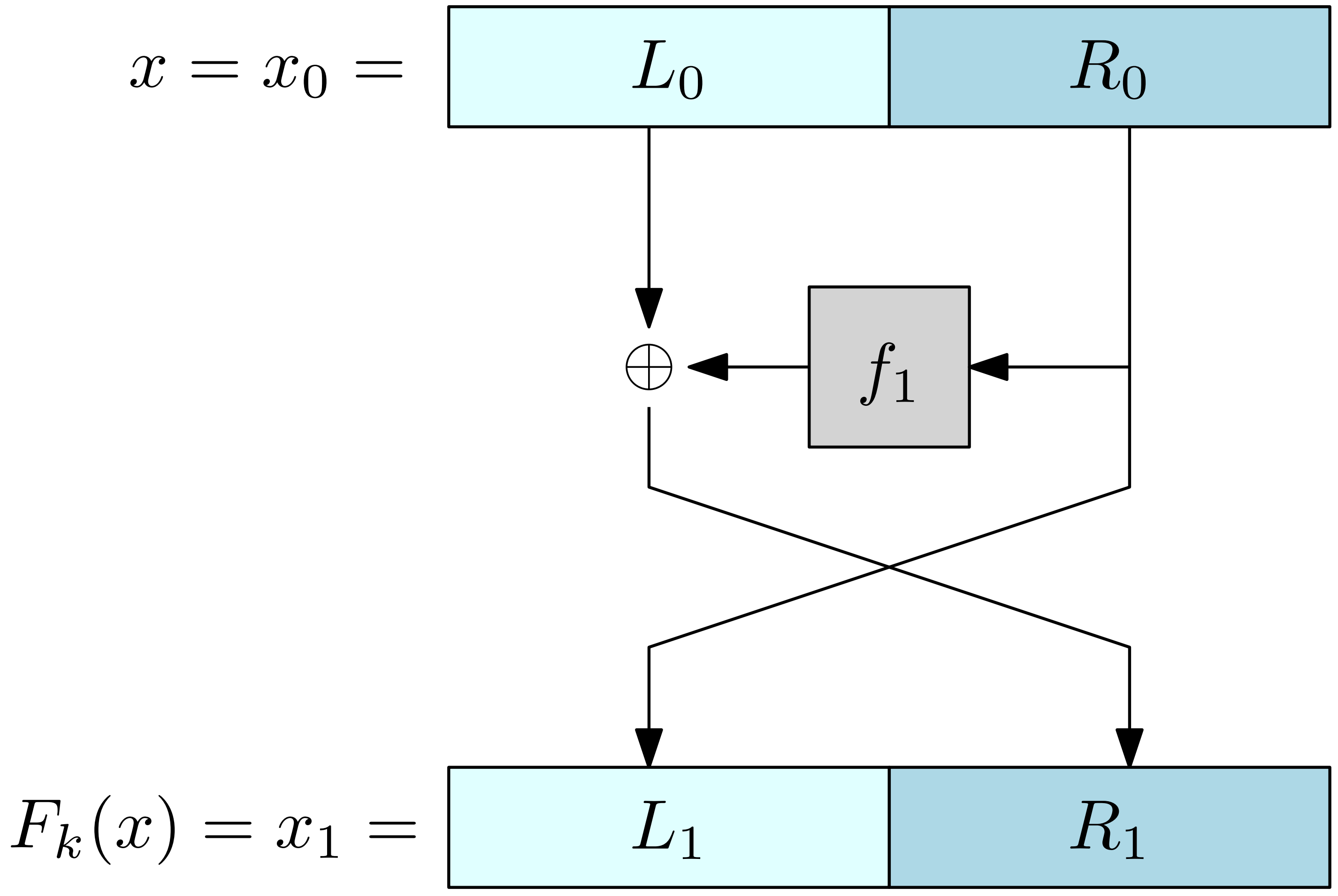
Security of 1-Round Feistel Networks

$F_k(L_0 \| R_0) = L_1 \| R_1$

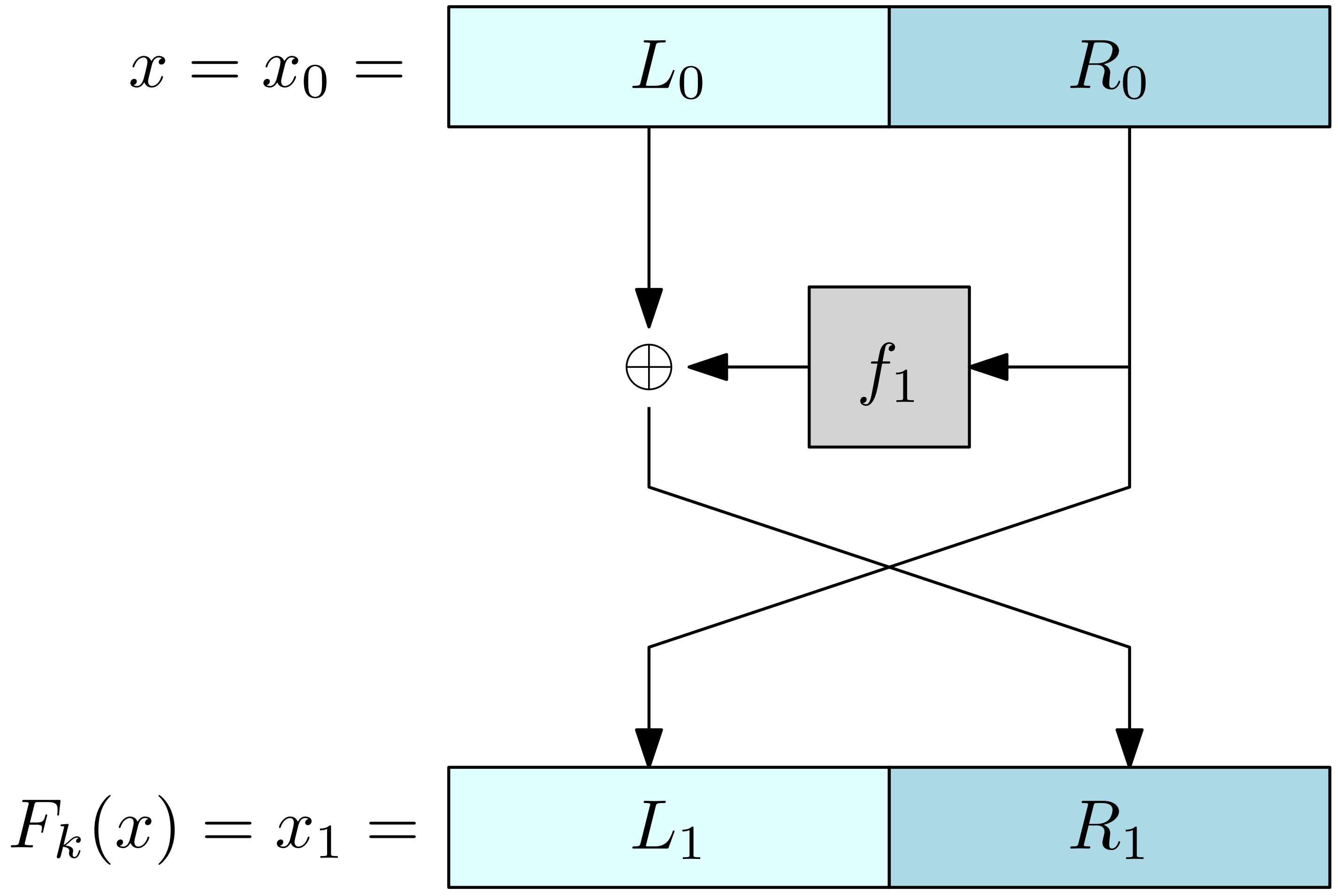
$L_1 = R_0$

$R_1 = L_0 \oplus f_1(R_0)$

Is this a Pseudorandom permutation?



Security of 1-Round Feistel Networks



$F_k(L_0 \,\|\, R_0) = L_1 \,\|\, R_1$

$L_1 = R_0$

$R_1 = L_0 \oplus f_1(R_0)$

Is this a Pseudorandom permutation?

- No! $F_k(x)$ can be easily distinguished from a random permutation

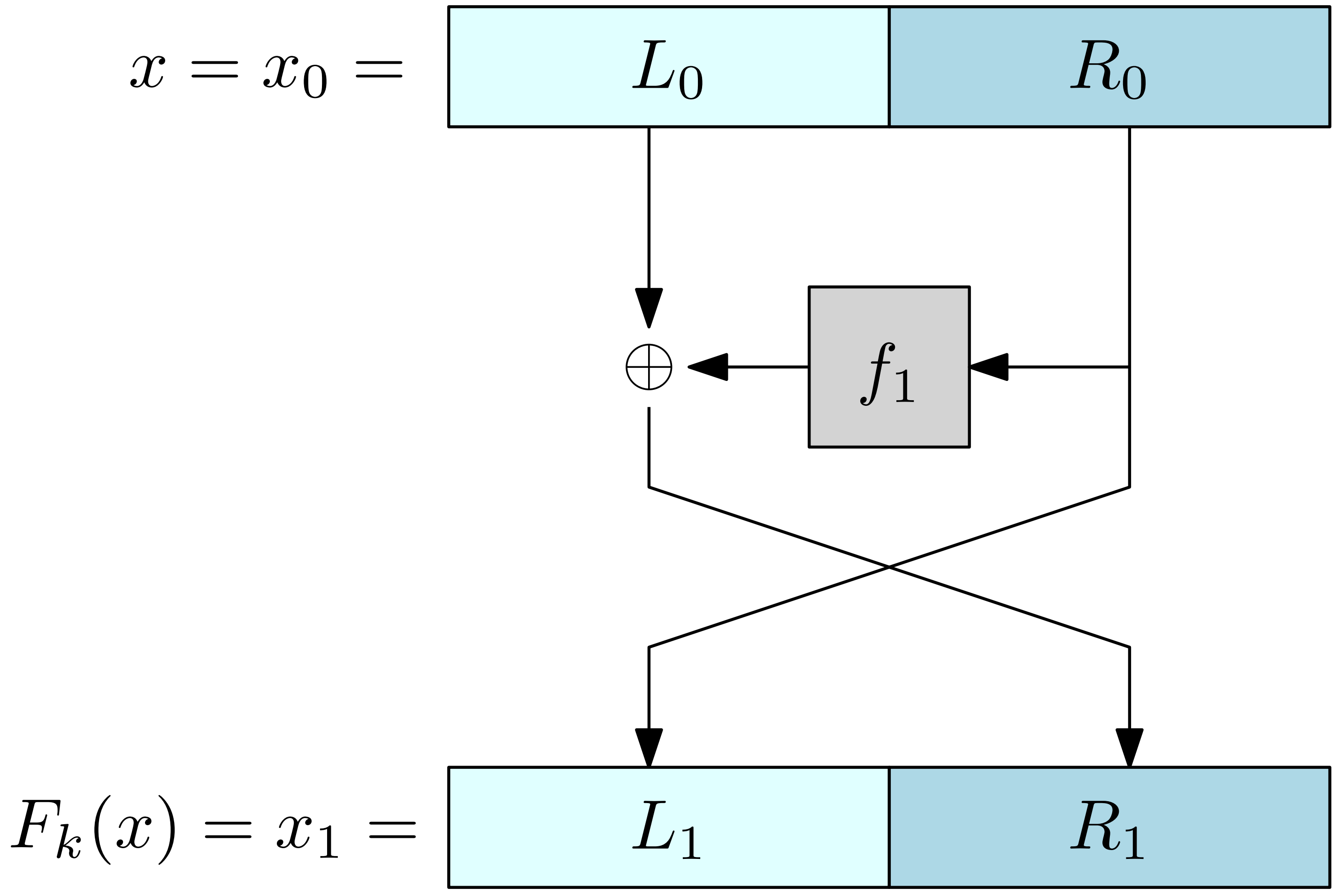
How?

Security of 1-Round Feistel Networks

$F_k(L_0 \| R_0) = L_1 \| R_1$

$L_1 = R_0$

$R_1 = L_0 \oplus f_1(R_0)$



Is this a Pseudorandom permutation?

- No! $F_k(x)$ can be easily distinguished from a random permutation

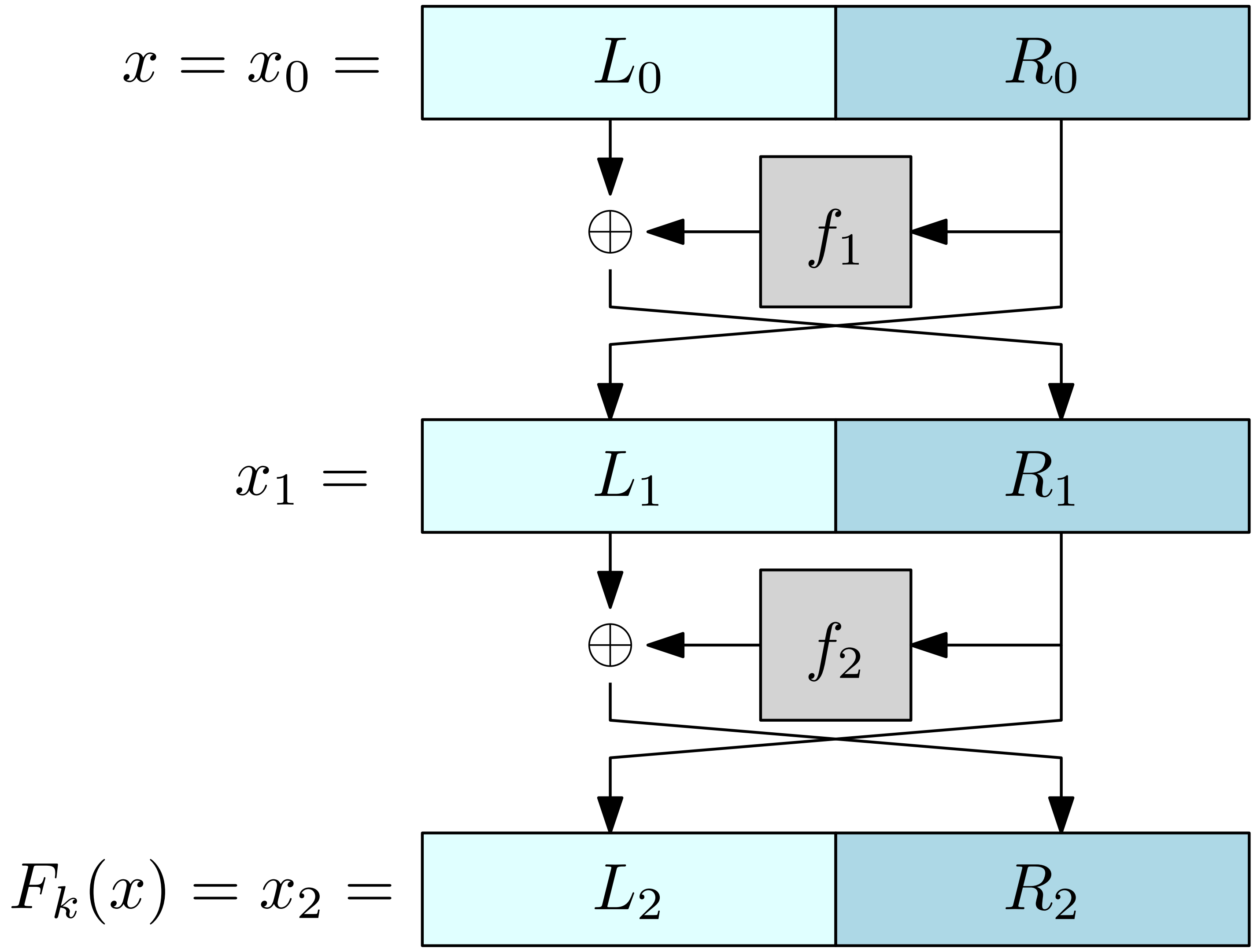
How?

- The adversary can simply query $x = 0^\ell$ and check whether the left $\ell/2$ bits of $F_k(x)$ are all 0

(or use any other string x and check whether the left half of $F_k(x)$ is equal to the right half of x)

Security of 2-Round Feistel Networks

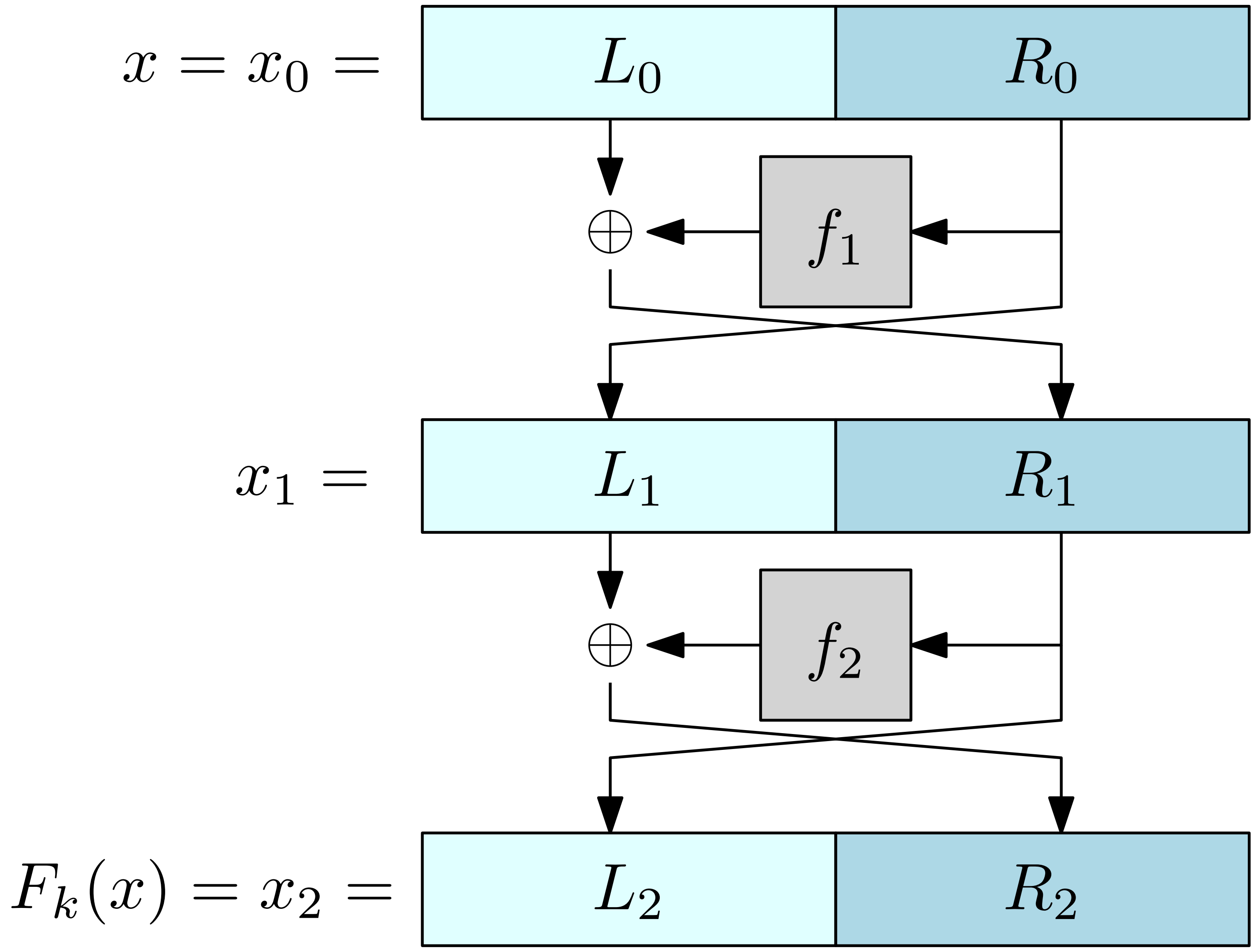
$F_k(L_0 \,\|\, R_0) = L_2 \,\|\, R_2$



Security of 2-Round Feistel Networks

$F_k(L_0 \,\|\, R_0) = L_2 \,\|\, R_2$

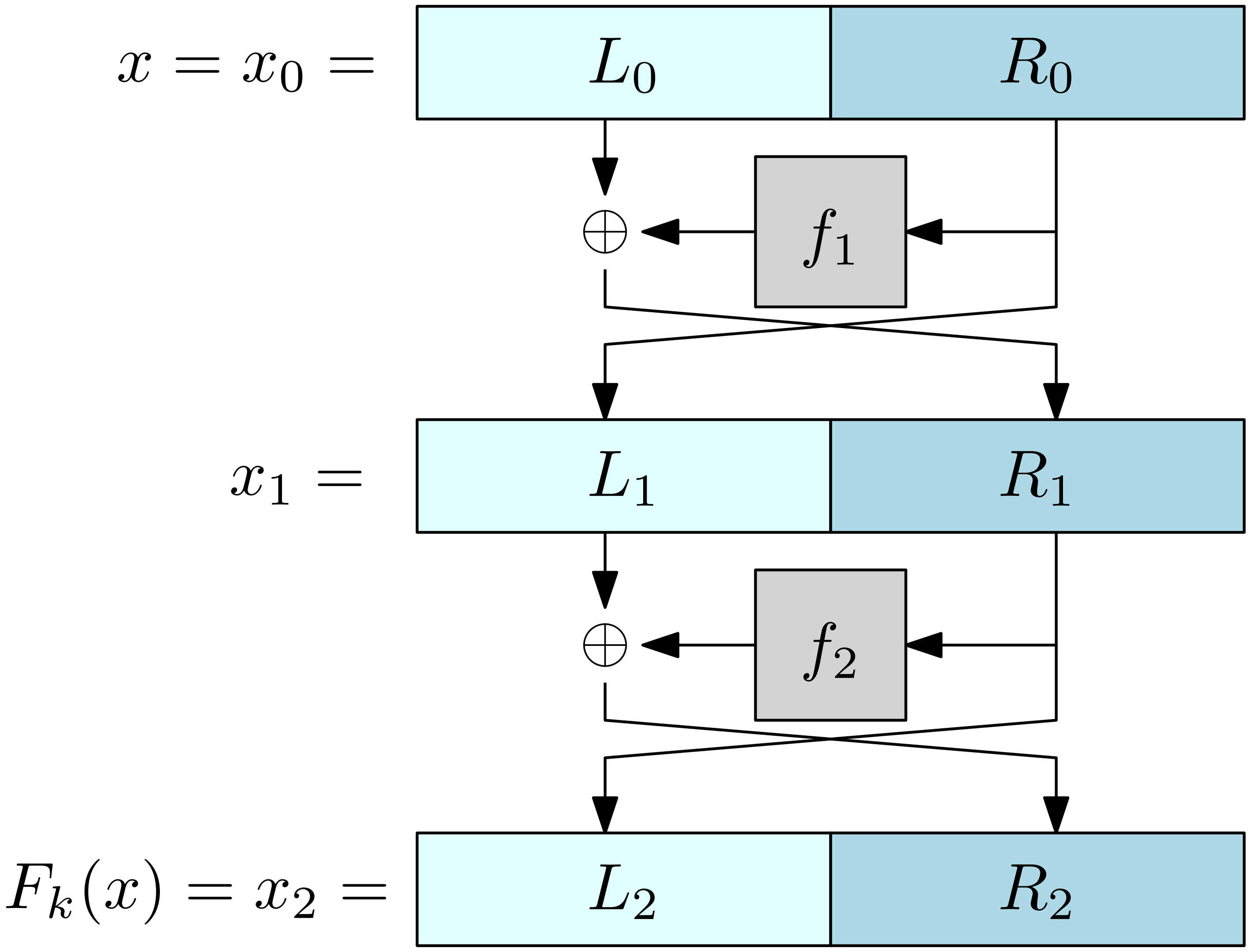
$L_2 = R_1$



Security of 2-Round Feistel Networks

$$F_k(L_0 \,\|\, R_0) = L_2 \,\|\, R_2$$

$$L_2 = R_1 = L_0 \oplus f_1(R_0)$$

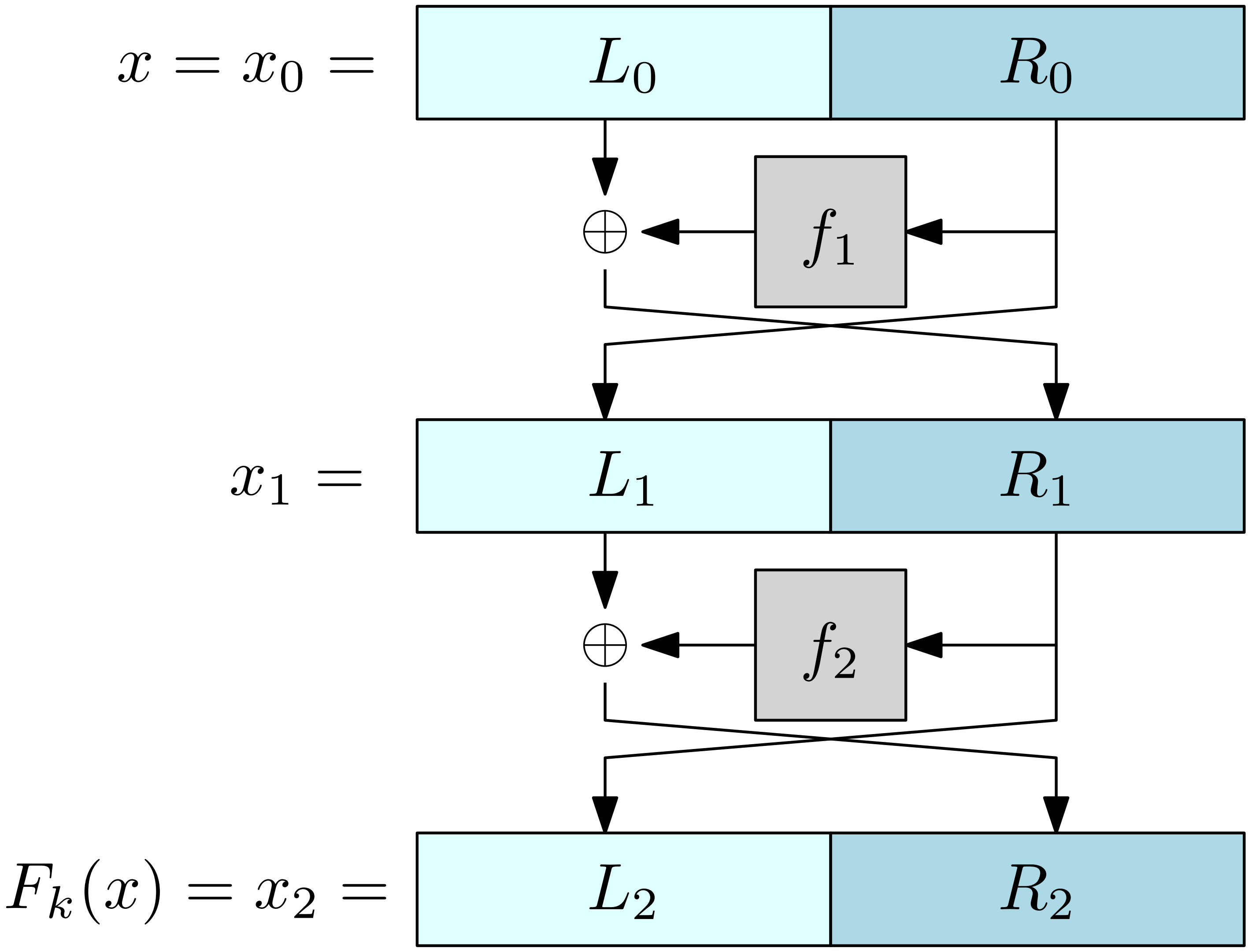


Security of 2-Round Feistel Networks

$F_k(L_0 \,\|\, R_0) = L_2 \,\|\, R_2$

$L_2 = R_1 = L_0 \oplus f_1(R_0)$

$R_2 = L_1 \oplus f_2(R_1)$

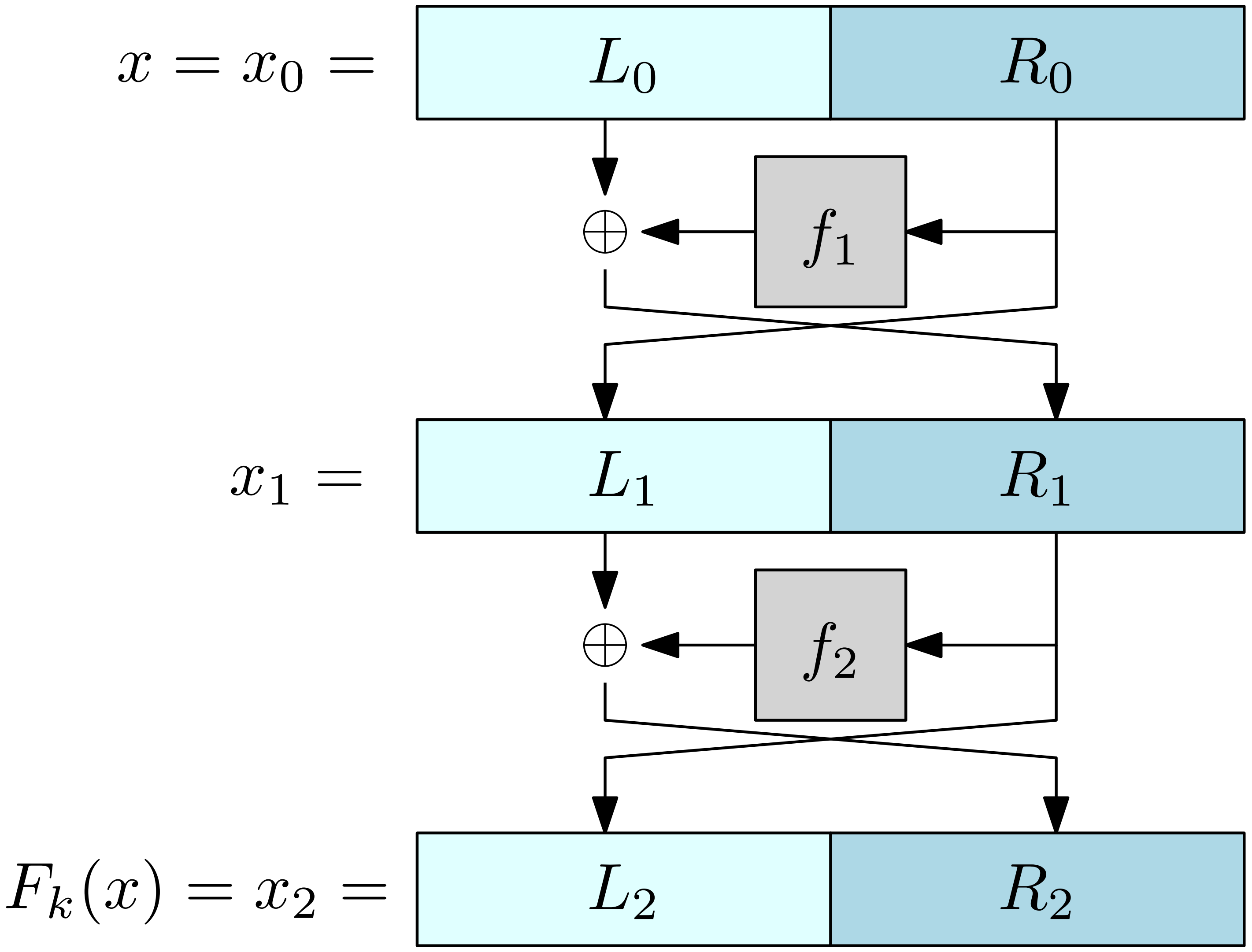


Security of 2-Round Feistel Networks

$$F_k(L_0 \,\|\, R_0) = L_2 \,\|\, R_2$$

$$L_2 = R_1 = L_0 \oplus f_1(R_0)$$

$$R_2 = L_1 \oplus f_2(R_1)$$
$$= R_0 \oplus f_2(R_1)$$

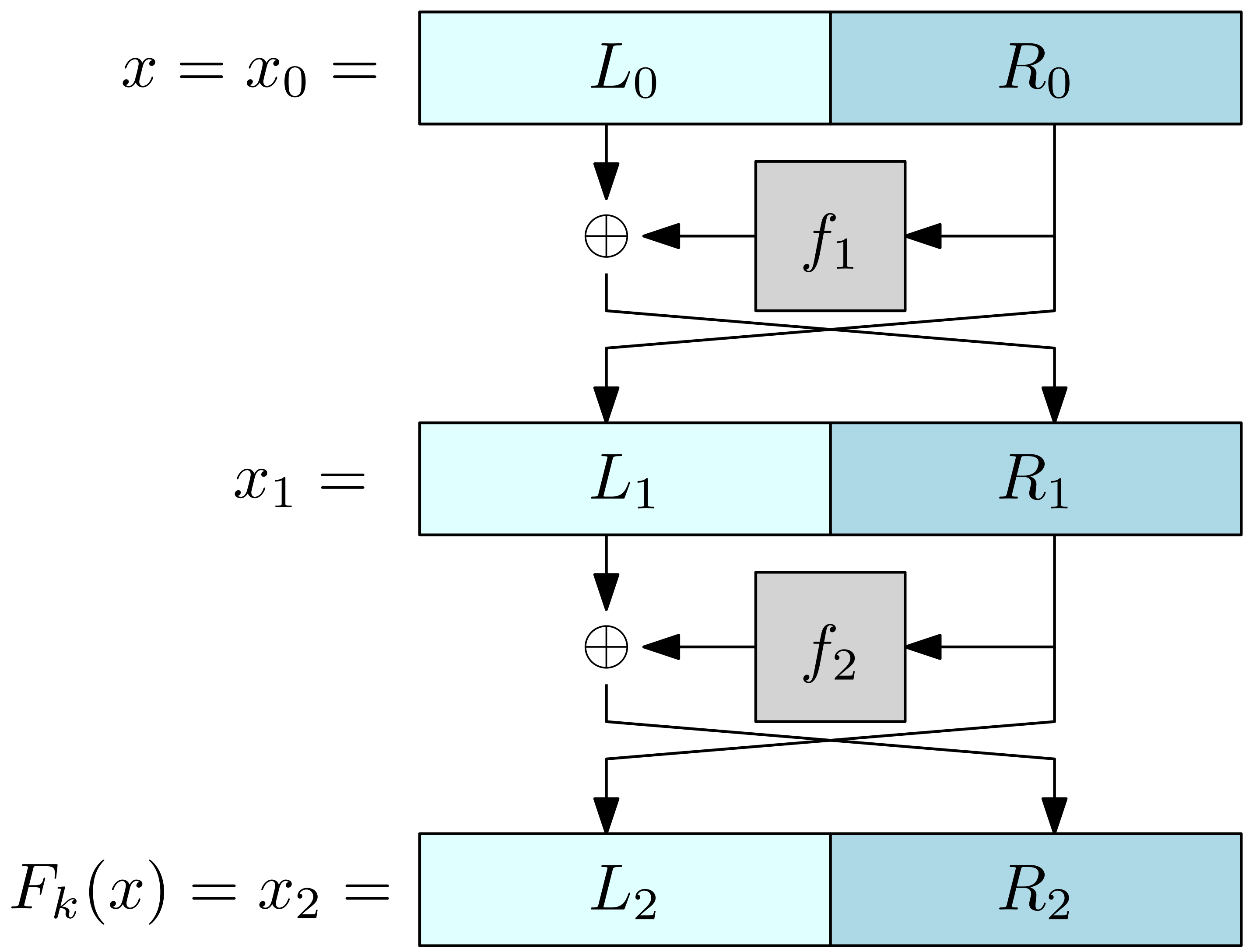


Security of 2-Round Feistel Networks

$F_k(L_0 \,\|\, R_0) = L_2 \,\|\, R_2$

$L_2 = R_1 = L_0 \oplus f_1(R_0)$

$$\begin{aligned} R_2 &= L_1 \oplus f_2(R_1) \\ &= R_0 \oplus f_2(R_1) \\ &= R_0 \oplus f_2(\, L_0 \oplus f_1(R_0)\,) \end{aligned}$$



Security of 2-Round Feistel Networks

$F_k(L_0 \| R_0) = L_2 \| R_2$

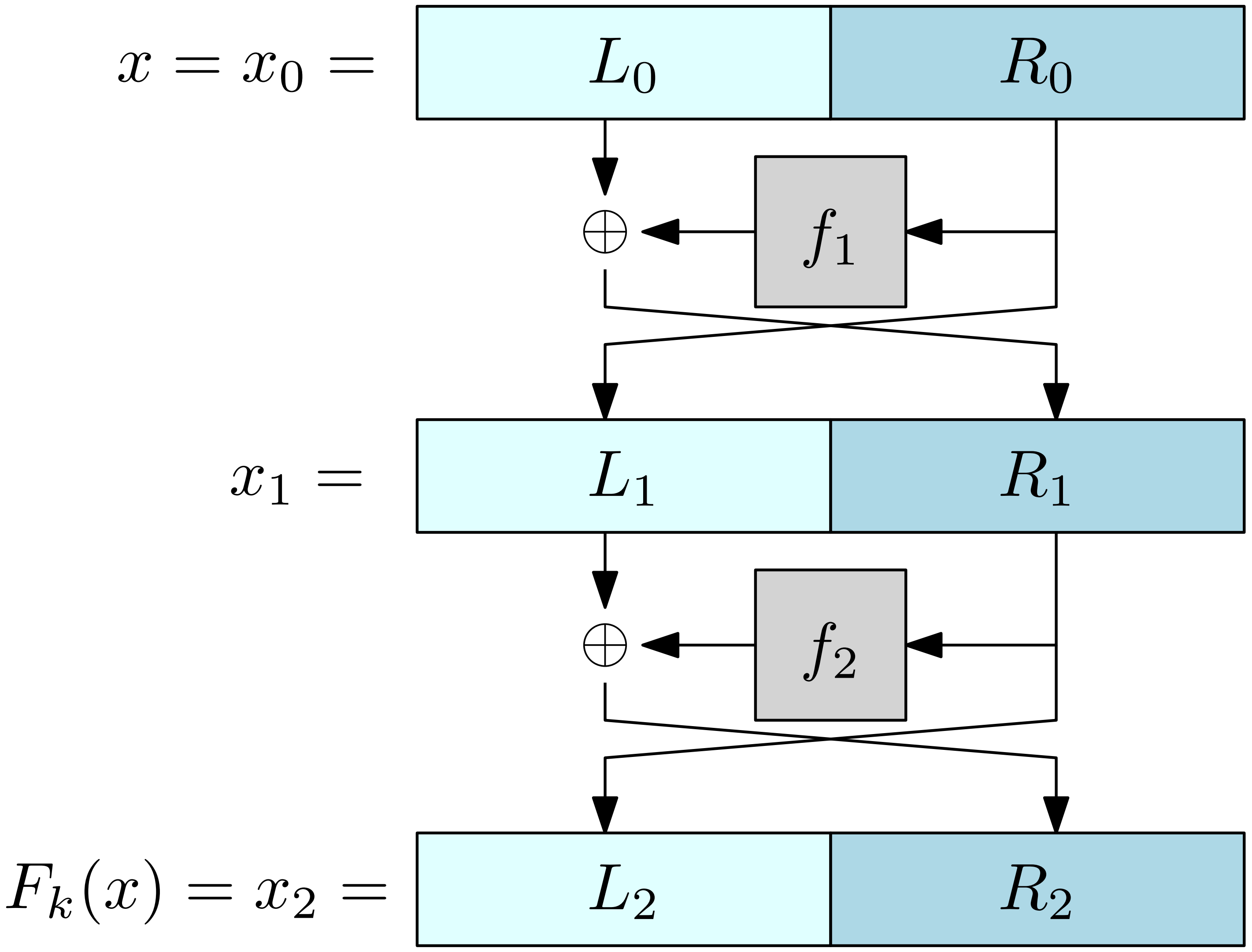
$L_2 = R_1 = L_0 \oplus f_1(R_0)$

$R_2 = L_1 \oplus f_2(R_1)$

$= R_0 \oplus f_2(R_1)$

$= R_0 \oplus f_2(\, L_0 \oplus f_1(R_0)\,)$

Is this a Pseudorandom permutation?



Security of 2-Round Feistel Networks

$F_k(L_0 \,\|\, R_0) = L_2 \,\|\, R_2$

$L_2 = R_1 = L_0 \oplus f_1(R_0)$

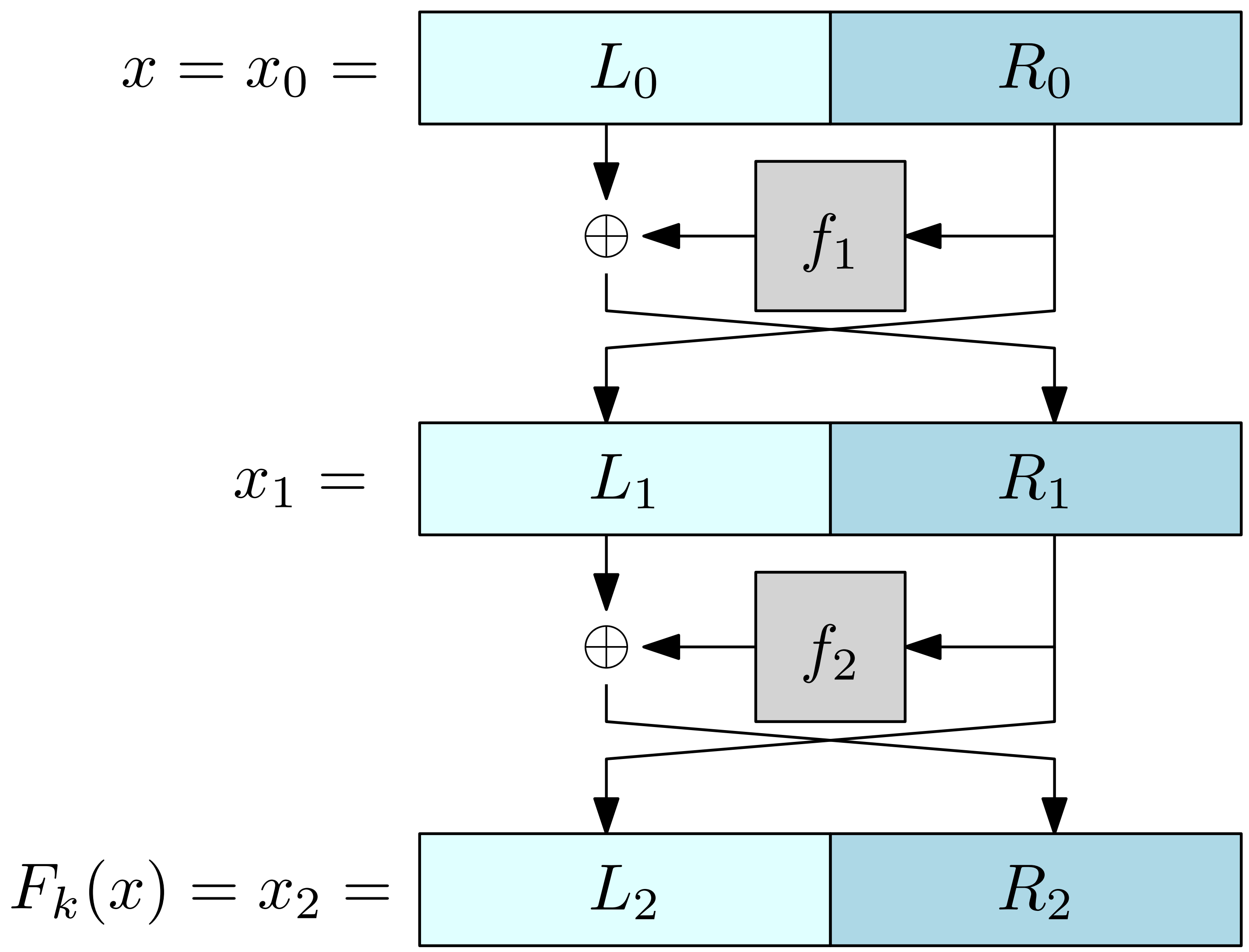
$R_2 = L_1 \oplus f_2(R_1)$

$= R_0 \oplus f_2(R_1)$

$= R_0 \oplus f_2(\, L_0 \oplus f_1(R_0)\,)$

Is this a Pseudorandom permutation?

No! Consider two different inputs $L_0 \| R_0$ and $L_0' \| R_0'$



Security of 2-Round Feistel Networks

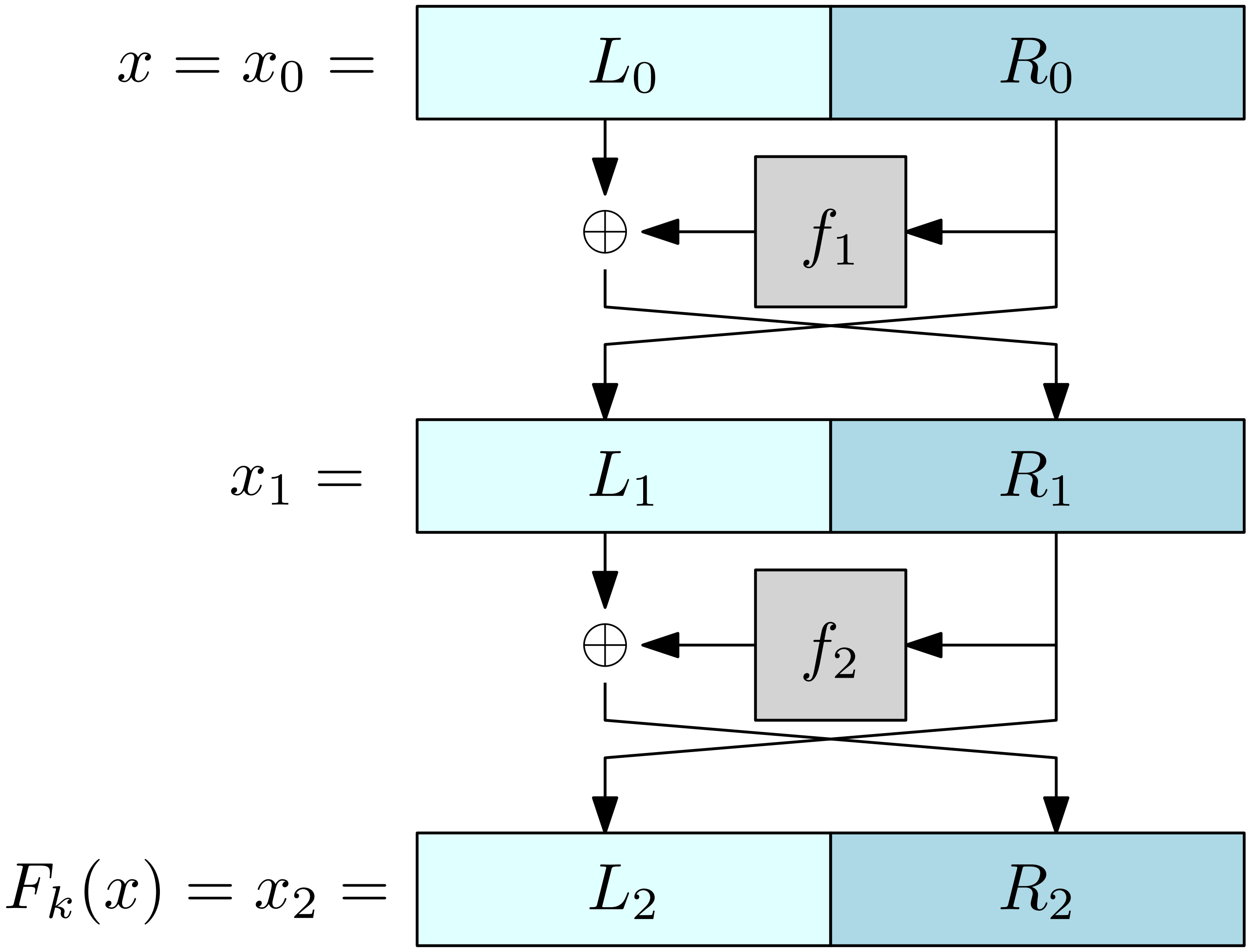
$F_k(L_0 \,\|\, R_0) = L_2 \,\|\, R_2$

$L_2 = R_1 = L_0 \oplus f_1(R_0)$

$R_2 = L_1 \oplus f_2(R_1)$

$= R_0 \oplus f_2(R_1)$

$= R_0 \oplus f_2(\, L_0 \oplus f_1(R_0)\,)$



Is this a Pseudorandom permutation?

No! Consider two different inputs $L_0 \| R_0$ and $L_0' \| R_0'$

$L_2 \oplus L_2'$

Security of 2-Round Feistel Networks

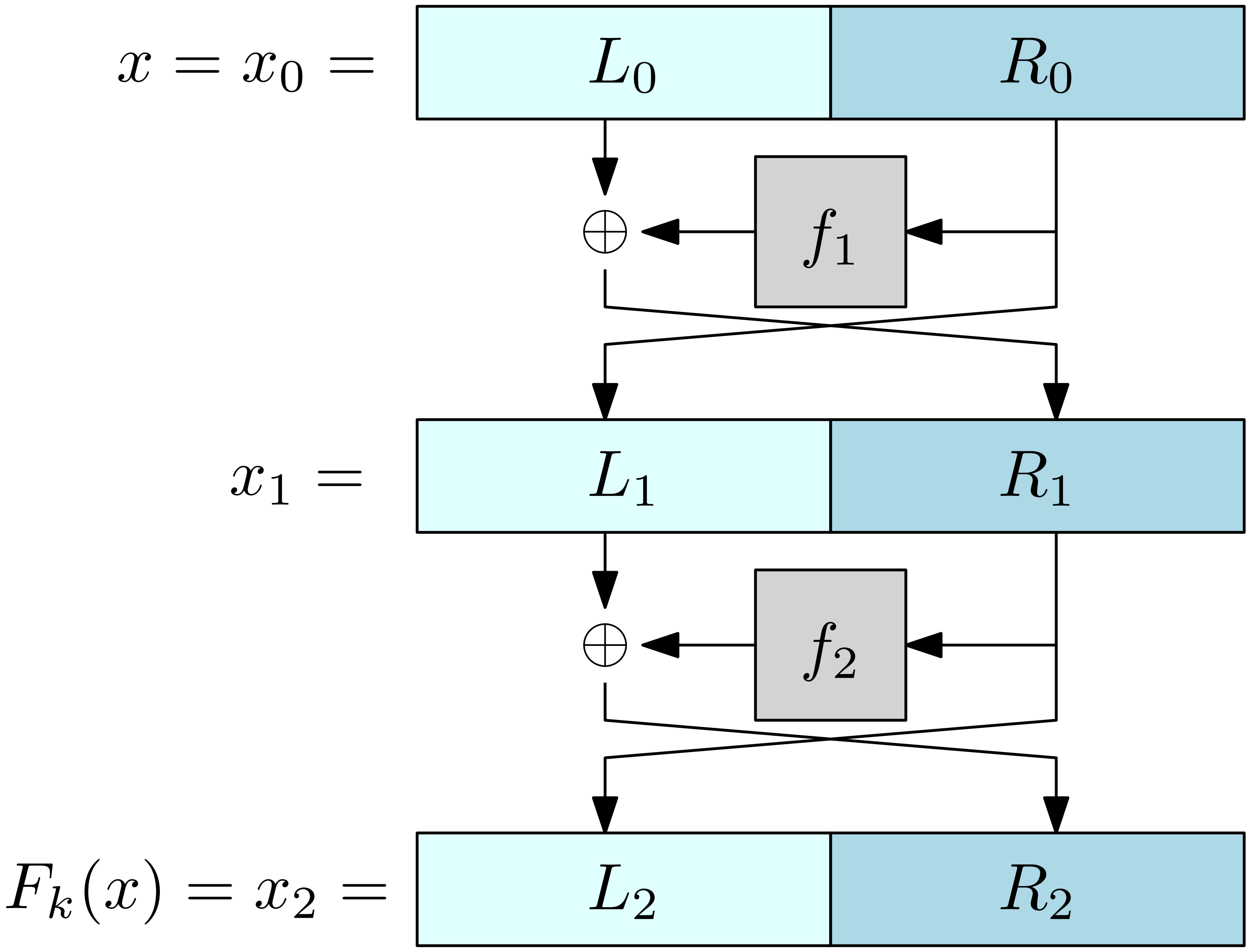
$F_k(L_0 \,\|\, R_0) = L_2 \,\|\, R_2$

$L_2 = R_1 = L_0 \oplus f_1(R_0)$

$R_2 = L_1 \oplus f_2(R_1)$

$\quad = R_0 \oplus f_2(R_1)$

$\quad = R_0 \oplus f_2(\, L_0 \oplus f_1(R_0)\,)$



Is this a Pseudorandom permutation?

No! Consider two different inputs
$L_0 \| R_0$ and $L'_0 \| R'_0$

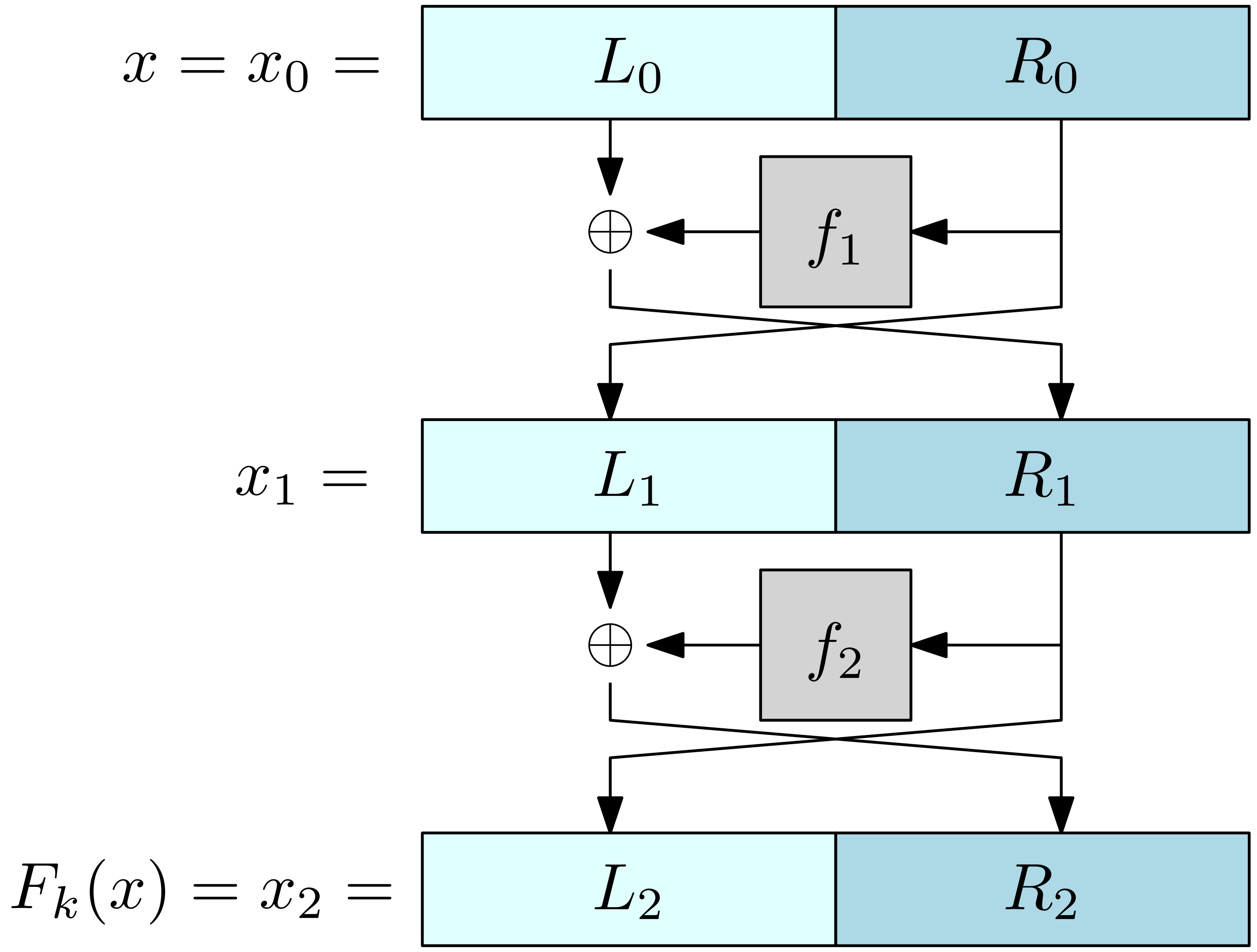
$L_2 \oplus L'_2 \; = L_0 \oplus f_1(R_0) \oplus L'_0 \oplus f_1(R'_0)$

Security of 2-Round Feistel Networks

$F_k(L_0 \,\|\, R_0) = L_2 \,\|\, R_2$

$L_2 = R_1 = L_0 \oplus f_1(R_0)$

$$
\begin{aligned}
R_2 &= L_1 \oplus f_2(R_1) \\
&= R_0 \oplus f_2(R_1) \\
&= R_0 \oplus f_2(\, L_0 \oplus f_1(R_0)\,)
\end{aligned}
$$



Is this a Pseudorandom permutation?

No! Consider two different inputs $L_0 \| R_0$ and $L'_0 \| R'_0$

$L_2 \oplus L'_2 \; = L_0 \oplus f_1(R_0) \oplus L'_0 \oplus f_1(R'_0)$

How can we exploit this?

Security of 2-Round Feistel Networks

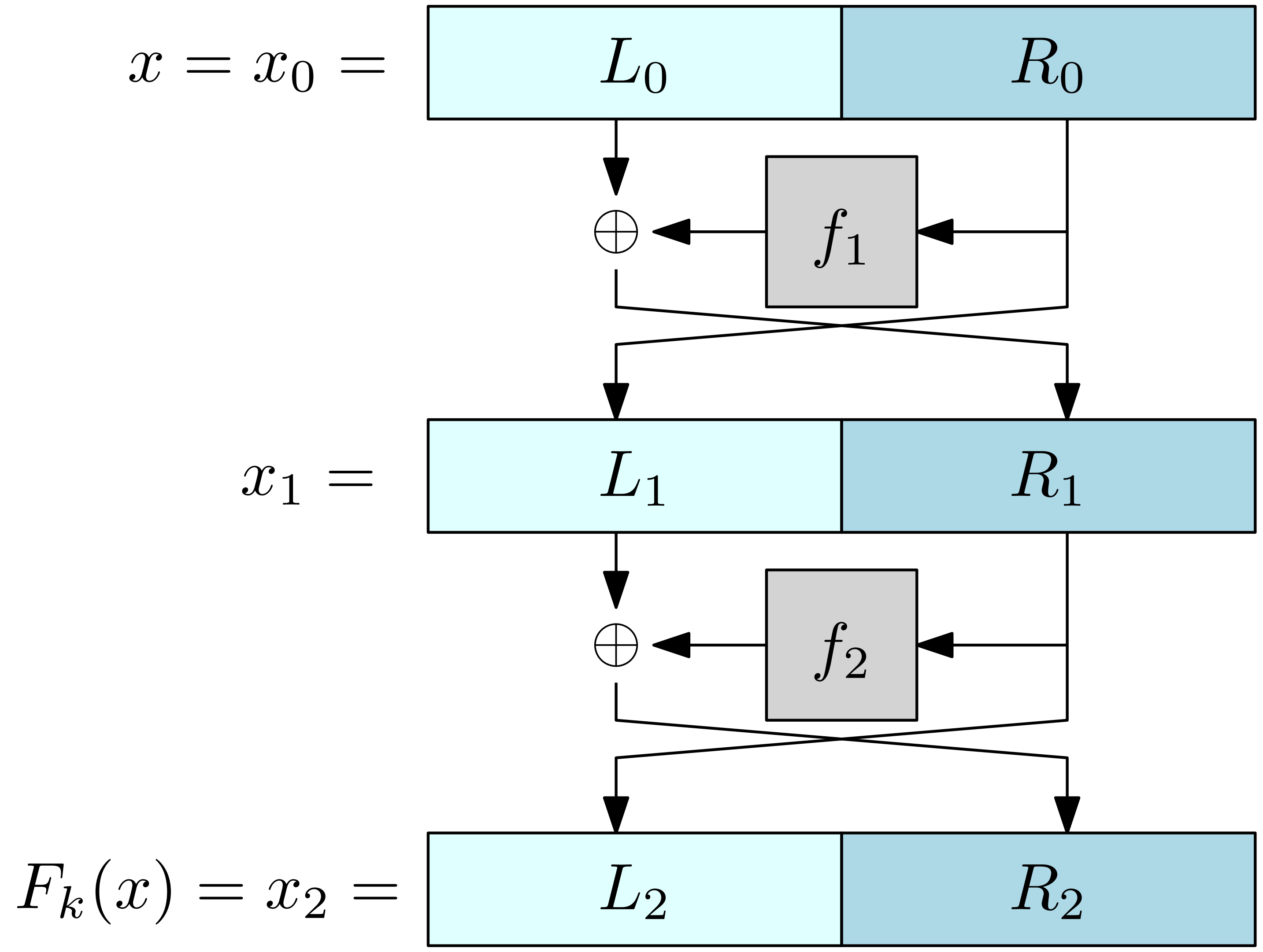
$F_k(L_0 \,\|\, R_0) = L_2 \,\|\, R_2$

$L_2 = R_1 = L_0 \oplus f_1(R_0)$

$R_2 = L_1 \oplus f_2(R_1)$

$= R_0 \oplus f_2(R_1)$

$= R_0 \oplus f_2(\, L_0 \oplus f_1(R_0)\,)$



Is this a Pseudorandom permutation?

No! Consider two different inputs $L_0 \| R_0$ and $L_0' \| R_0'$

$L_2 \oplus L_2' = L_0 \oplus f_1(R_0) \oplus L_0' \oplus f_1(R_0')$

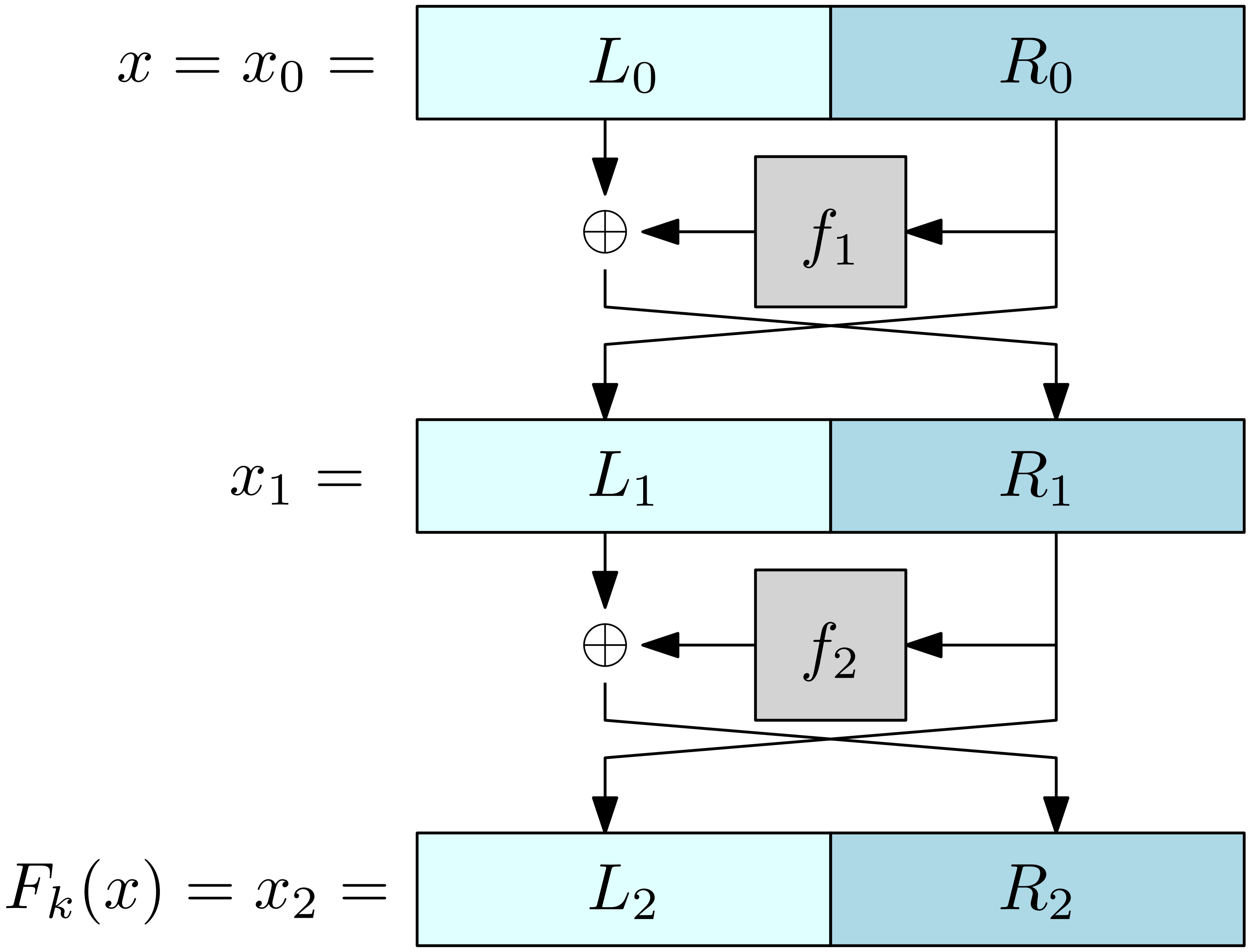
How can we exploit this? Pick $R_0 = R_0'$

Security of 2-Round Feistel Networks

$F_k(L_0 \,\|\, R_0) = L_2 \,\|\, R_2$

$L_2 = R_1 = L_0 \oplus f_1(R_0)$

$$\begin{aligned} R_2 &= L_1 \oplus f_2(R_1) \\ &= R_0 \oplus f_2(R_1) \\ &= R_0 \oplus f_2(\, L_0 \oplus f_1(R_0)\,) \end{aligned}$$



Is this a Pseudorandom permutation?

No! Consider two different inputs $L_0 \| R_0$ and $L_0' \| R_0'$

$$\begin{aligned} L_2 \oplus L_2' &= L_0 \oplus f_1(R_0) \oplus L_0' \oplus f_1(R_0') \\ &= L_0 \oplus L_0' \end{aligned}$$

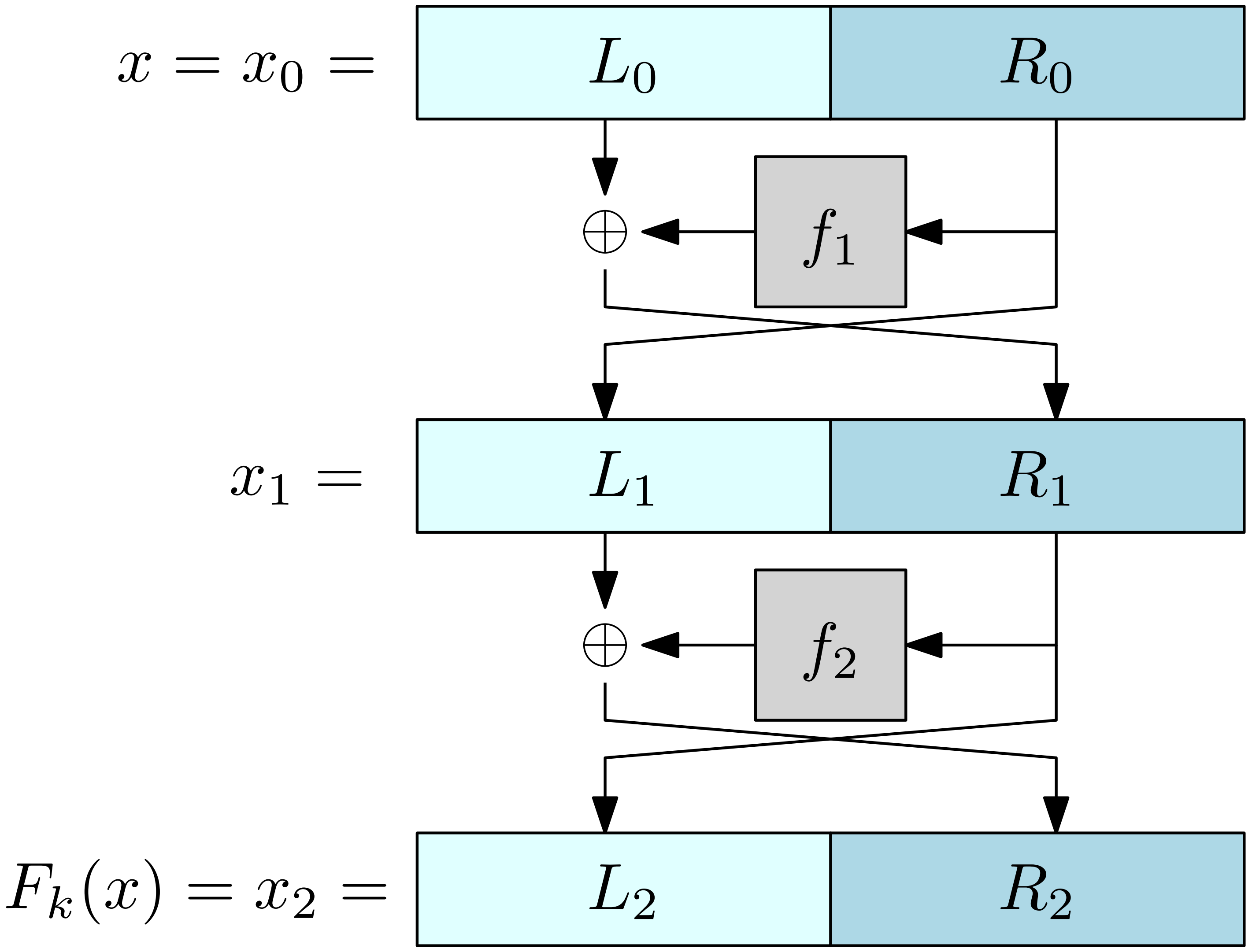
How can we exploit this? Pick $R_0 = R_0'$

Security of 2-Round Feistel Networks

$F_k(L_0 \| R_0) = L_2 \| R_2$

$L_2 = R_1 = L_0 \oplus f_1(R_0)$

$$R_2 = L_1 \oplus f_2(R_1)$$
$$= R_0 \oplus f_2(R_1)$$
$$= R_0 \oplus f_2(\, L_0 \oplus f_1(R_0)\,)$$



Is this a Pseudorandom permutation?

No! Consider two different inputs $L_0 \| R_0$ and $L'_0 \| R'_0$

$$L_2 \oplus L'_2 = L_0 \oplus f_1(R_0) \oplus L'_0 \oplus f_1(R'_0)$$
$$= L_0 \oplus L'_0$$

How can we exploit this? Pick $R_0 = R'_0$

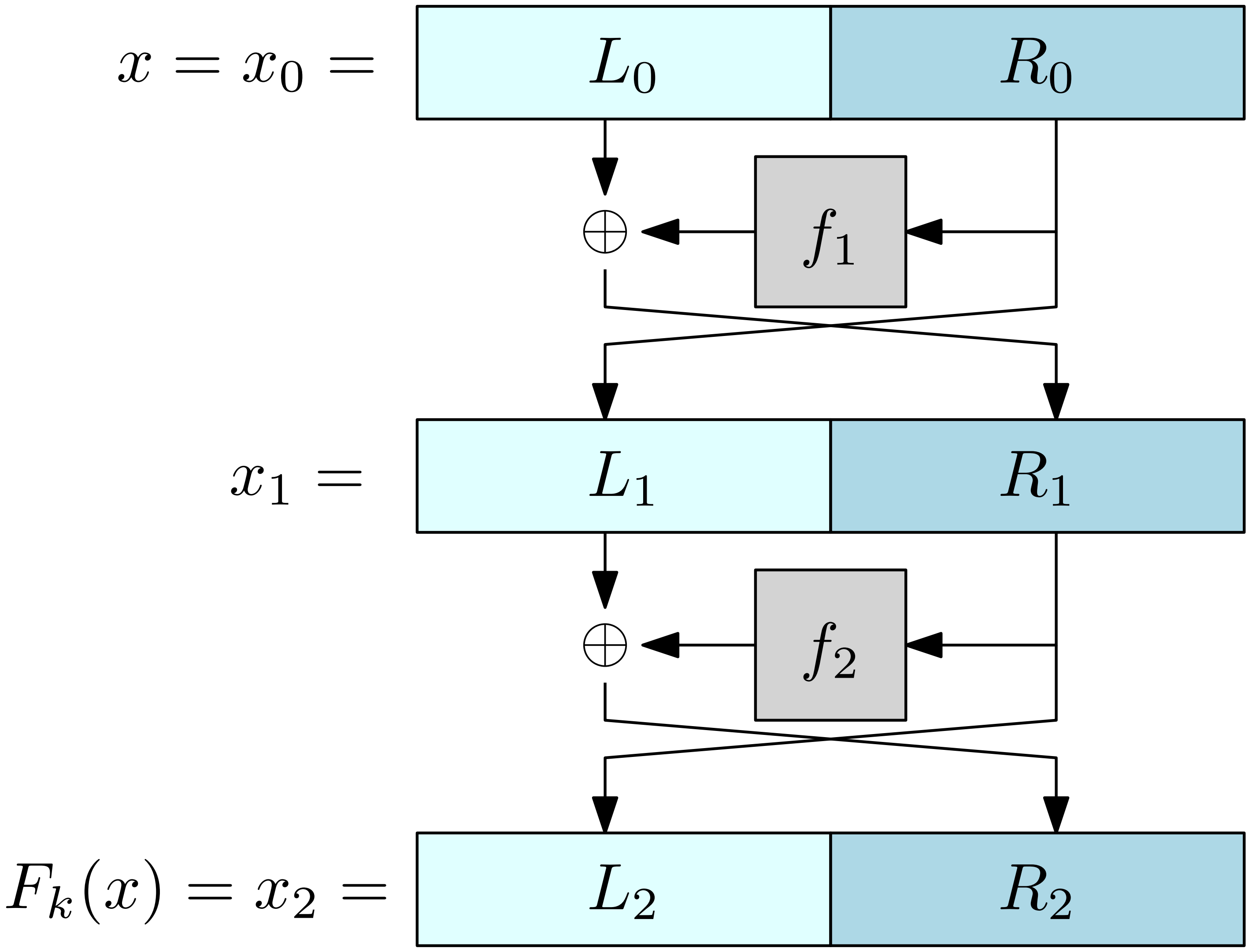
This is easy to distinguish from a random function. E.g., pick $L_0 = 0^{\ell/2}$ and $L'_0 = 1^{\ell/2}$

Security of 2-Round Feistel Networks

$F_k(L_0 \,\|\, R_0) = L_2 \,\|\, R_2$

$L_2 = R_1 = L_0 \oplus f_1(R_0)$

$$R_2 = L_1 \oplus f_2(R_1)$$
$$= R_0 \oplus f_2(R_1)$$
$$= R_0 \oplus f_2(\, L_0 \oplus f_1(R_0)\,)$$



Is this a Pseudorandom permutation?

No! Consider two different inputs $L_0 \| R_0$ and $L'_0 \| R'_0$

$$L_2 \oplus L'_2 = L_0 \oplus f_1(R_0) \oplus L'_0 \oplus f_1(R'_0)$$
$$= L_0 \oplus L'_0$$
$$= 0^{\ell/2} \oplus 1^{\ell/2}$$

How can we exploit this? Pick $R_0 = R'_0$

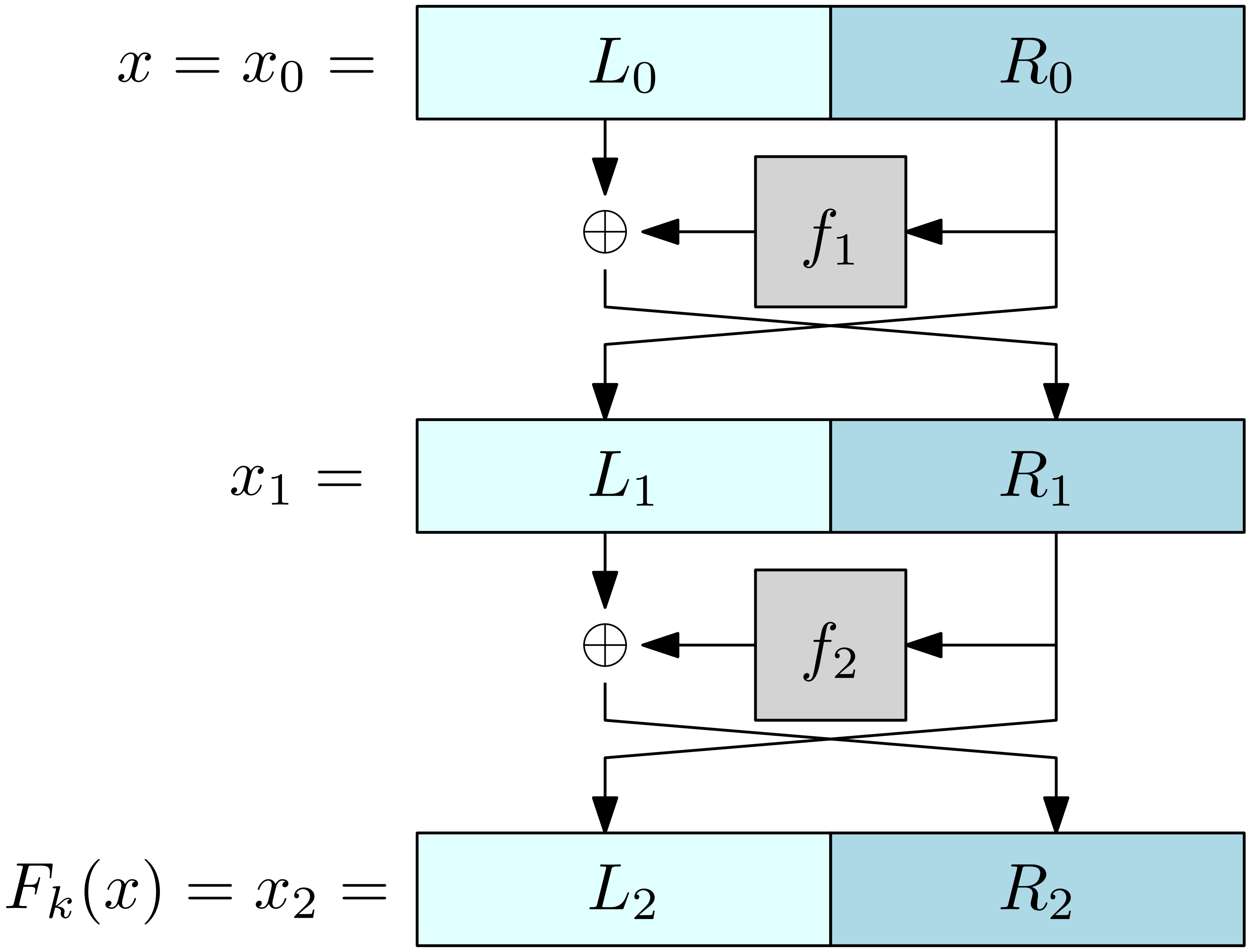
This is easy to distinguish from a random function. E.g., pick $L_0 = 0^{\ell/2}$ and $L'_0 = 1^{\ell/2}$

Security of 2-Round Feistel Networks

$F_k(L_0 \| R_0) = L_2 \| R_2$

$L_2 = R_1 = L_0 \oplus f_1(R_0)$

$$R_2 = L_1 \oplus f_2(R_1) = R_0 \oplus f_2(R_1) = R_0 \oplus f_2(L_0 \oplus f_1(R_0))$$



Is this a Pseudorandom permutation?

No! Consider two different inputs $L_0 \| R_0$ and $L'_0 \| R'_0$

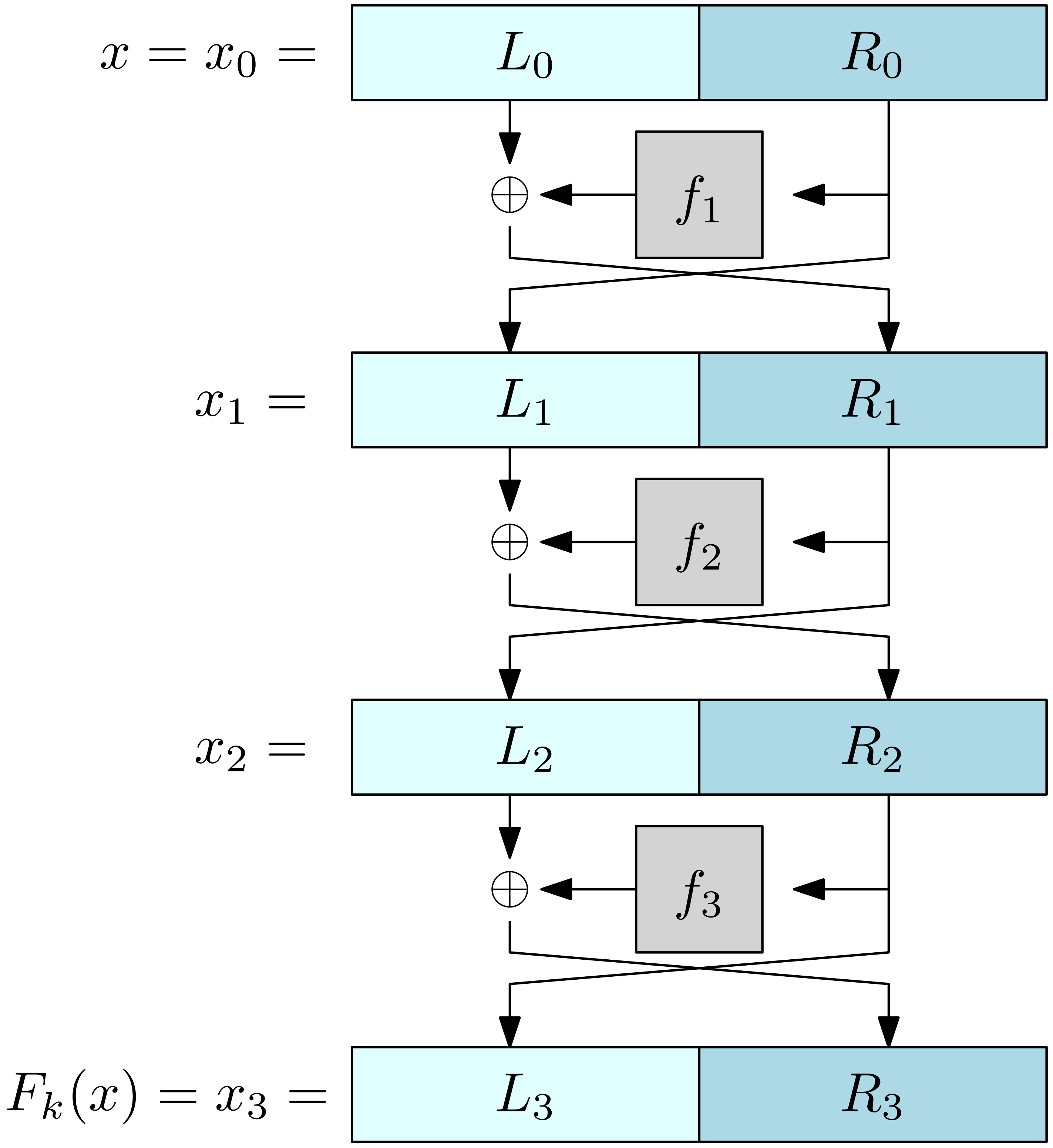
$$L_2 \oplus L'_2 = L_0 \oplus f_1(R_0) \oplus L'_0 \oplus f_1(R'_0) = L_0 \oplus L'_0 = 0^{\ell/2} \oplus 1^{\ell/2} = 1^{\ell/2}$$

How can we exploit this? Pick $R_0 = R'_0$

This is easy to distinguish from a random function. E.g., pick $L_0 = 0^{\ell/2}$ and $L'_0 = 1^{\ell/2}$

Security of 3-Round Feistel Networks

Is this a pseudorandom permutation?

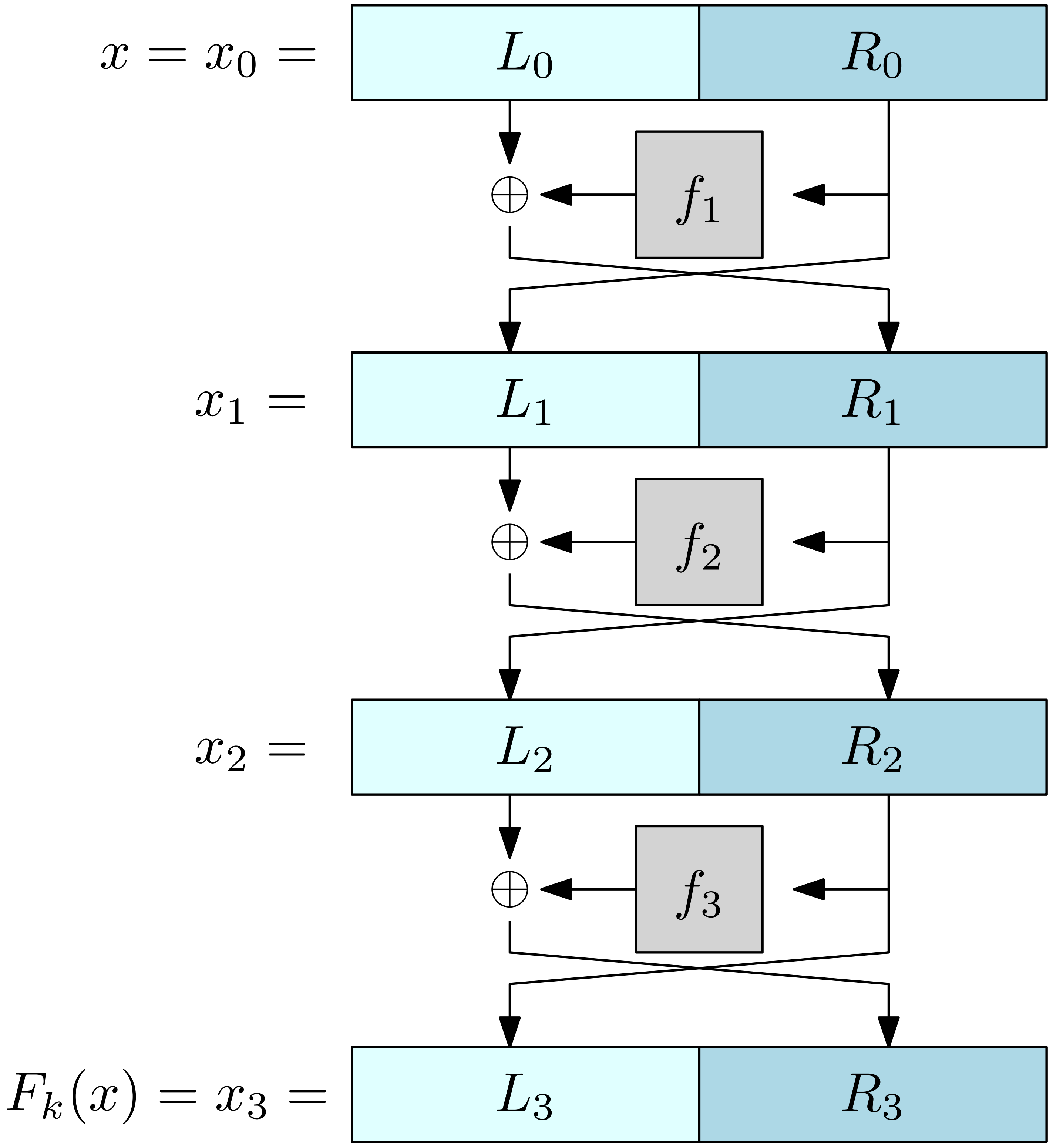


Security of 3-Round Feistel Networks

Is this a pseudorandom permutation?

- Yes!

  (If $f_i = F_{k_i}$ for some pseudorandom function F and the keys k_i are chosen independently and u.a.r.)



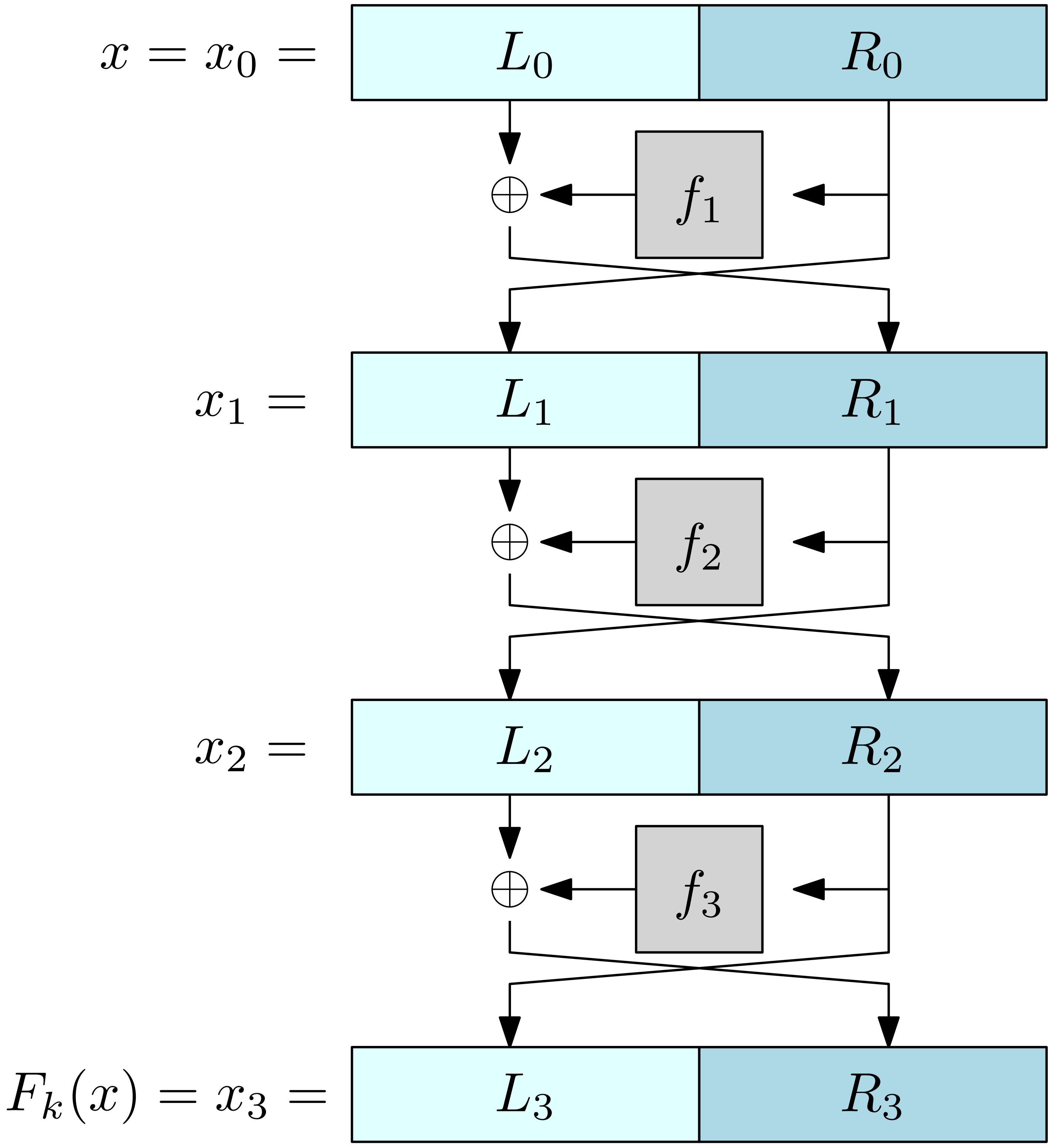
Security of 3-Round Feistel Networks

Is this a pseudorandom permutation?

- Yes!

  (If $f_i = F_{k_i}$ for some pseudorandom function F and the keys k_i are chosen independently and u.a.r.)

Is this a **strong** pseudorandom permutation?



Security of 3-Round Feistel Networks

Is this a pseudorandom permutation?

- Yes!

  (If $f_i = F_{k_i}$ for some pseudorandom function F and the keys k_i are chosen independently and u.a.r.)

Is this a **strong** pseudorandom permutation?

- No
- But 4-round Feistel networks are!