

CMSC/Math 456: Cryptography (Fall 2022)

Lecture 5

Daniel Gottesman

Administrative

The second problem set is out, due next Tuesday (Sep. 20) at noon, again on Gradescope.

Use of Piazza:

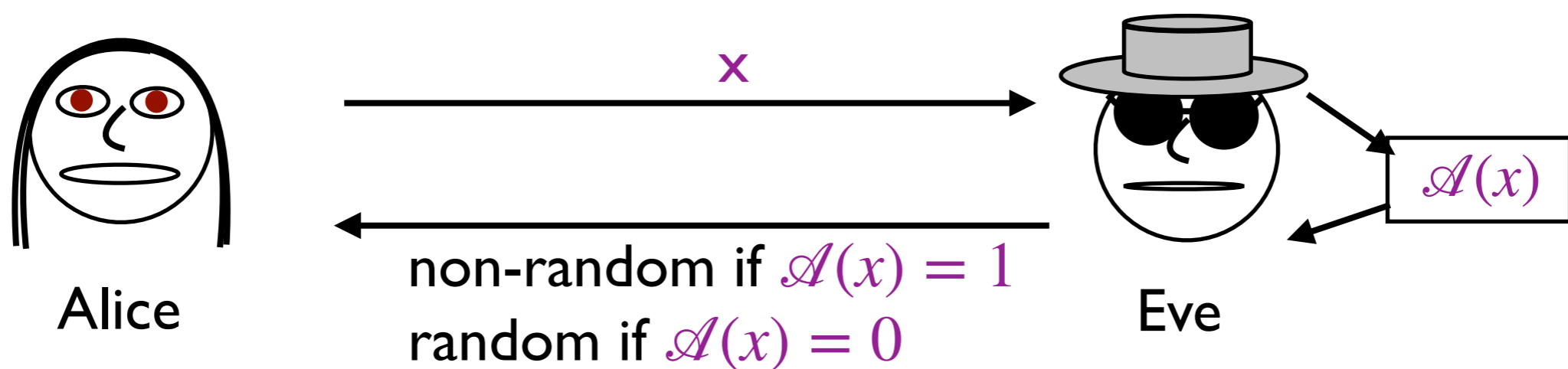
I have received a few questions on the HW asked privately on Piazza and have so far left them private. But unless your question is about your personal situation or reveals part of the answer to the homework, please ask publicly (which could be anonymous, if you choose) so that other students can benefit from the answer. In the future, I may switch them to public.

Pseudorandom Generators

Definition: Let $G : \{0,1\}^s \rightarrow \{0,1\}^{\ell(s)}$ be a *deterministic efficient* function with $\ell(s) > s$ for all s . Then $G(y)$ is a **pseudorandom generator** if, for any attack $\mathcal{A} : \{0,1\}^{\ell(s)} \rightarrow \{0,1\}$, a probabilistic polynomial time algorithm, it holds that

$$|\Pr_y(\mathcal{A}(G(y)) = 1) - \Pr_x(\mathcal{A}(x) = 1)| \leq \epsilon(s)$$

with $\epsilon(s)$ a negligible function and probabilities averaged over randomness of \mathcal{A} , as well as over **seeds** y (left probability) drawn uniformly from $\{0,1\}^s$ and truly random strings x (right probability) drawn uniformly from $\{0,1\}^{\ell(s)}$.

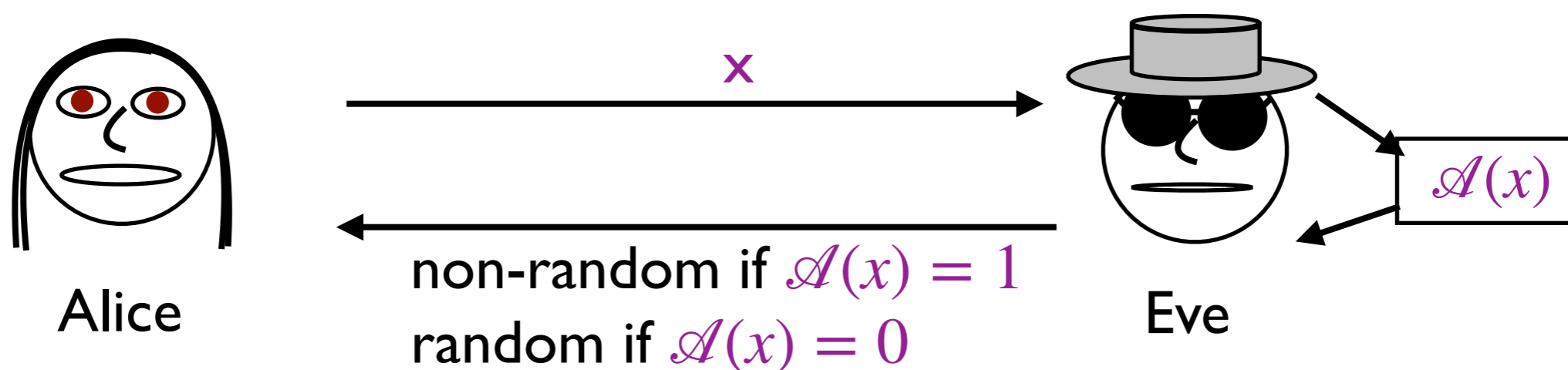


Pseudorandom Generators

Definition: Let $G : \{0,1\}^s \rightarrow \{0,1\}^{\ell(s)}$ be a *deterministic* **efficient** function with $\ell(s) > s$ for all s . Then $G(y)$ is a **pseudorandom generator** if, for any attack $\mathcal{A} : \{0,1\}^{\ell(s)} \rightarrow \{0,1\}$, a probabilistic polynomial time algorithm, it holds that

$$|\Pr_y(\mathcal{A}(G(y)) = 1) - \Pr_x(\mathcal{A}(x) = 1)| \leq \epsilon(s)$$

with $\epsilon(s)$ a negligible function and probabilities averaged over randomness of \mathcal{A} , as well as over **seeds** y (left probability) drawn uniformly from $\{0,1\}^s$ and truly random strings x (right probability) drawn uniformly from $\{0,1\}^{\ell(s)}$.

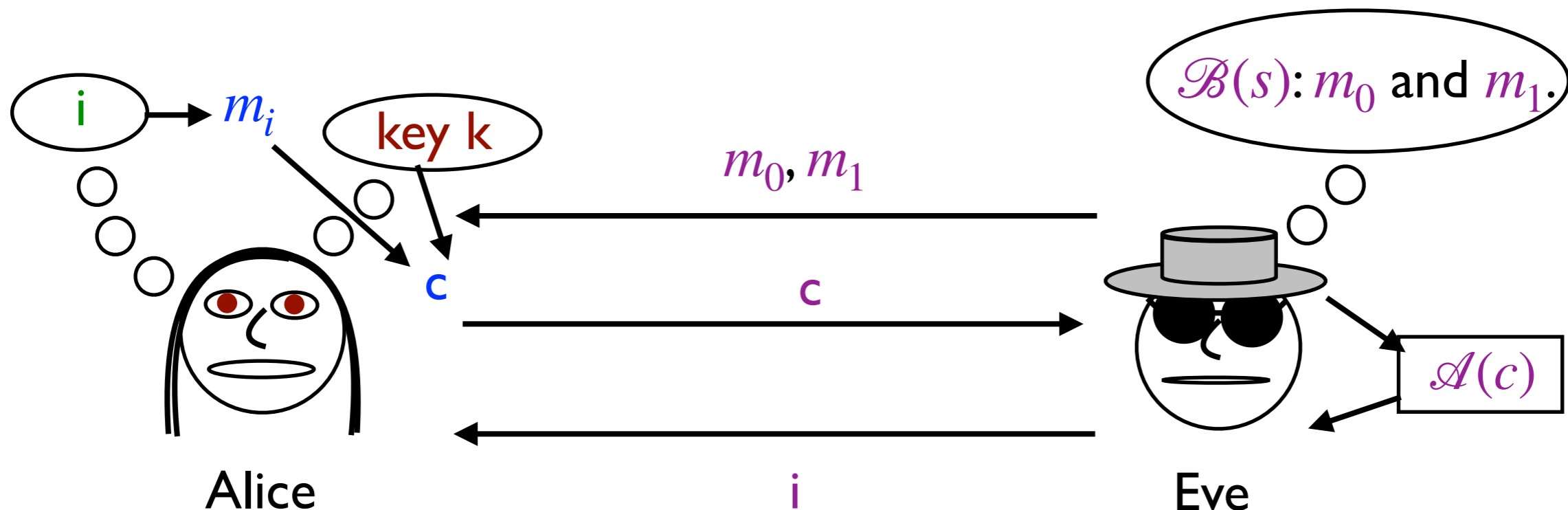


EAV Security

Definition: (Enc, Dec) with security parameter s has indistinguishable encryptions in the presence of an eavesdropper (is EAV-secure) if, for any pair of messages m_0 and m_1 chosen by the adversary (using $\mathcal{B}(s)$) and for any efficient attack $\mathcal{A}(c)$,

$$|\Pr_k(\mathcal{A}(\text{Enc}(k, m_0)) = 1) - \Pr_k(\mathcal{A}(\text{Enc}(k, m_1)) = 1)| \leq \epsilon(s)$$

for negligible $\epsilon(s)$ and probability taken over k and randomness of Enc .



Security of Pseudo One-Time Pad

Theorem: The pseudo one-time pad is EAV-secure if it uses a secure pseudorandom generator.

Proof plan:

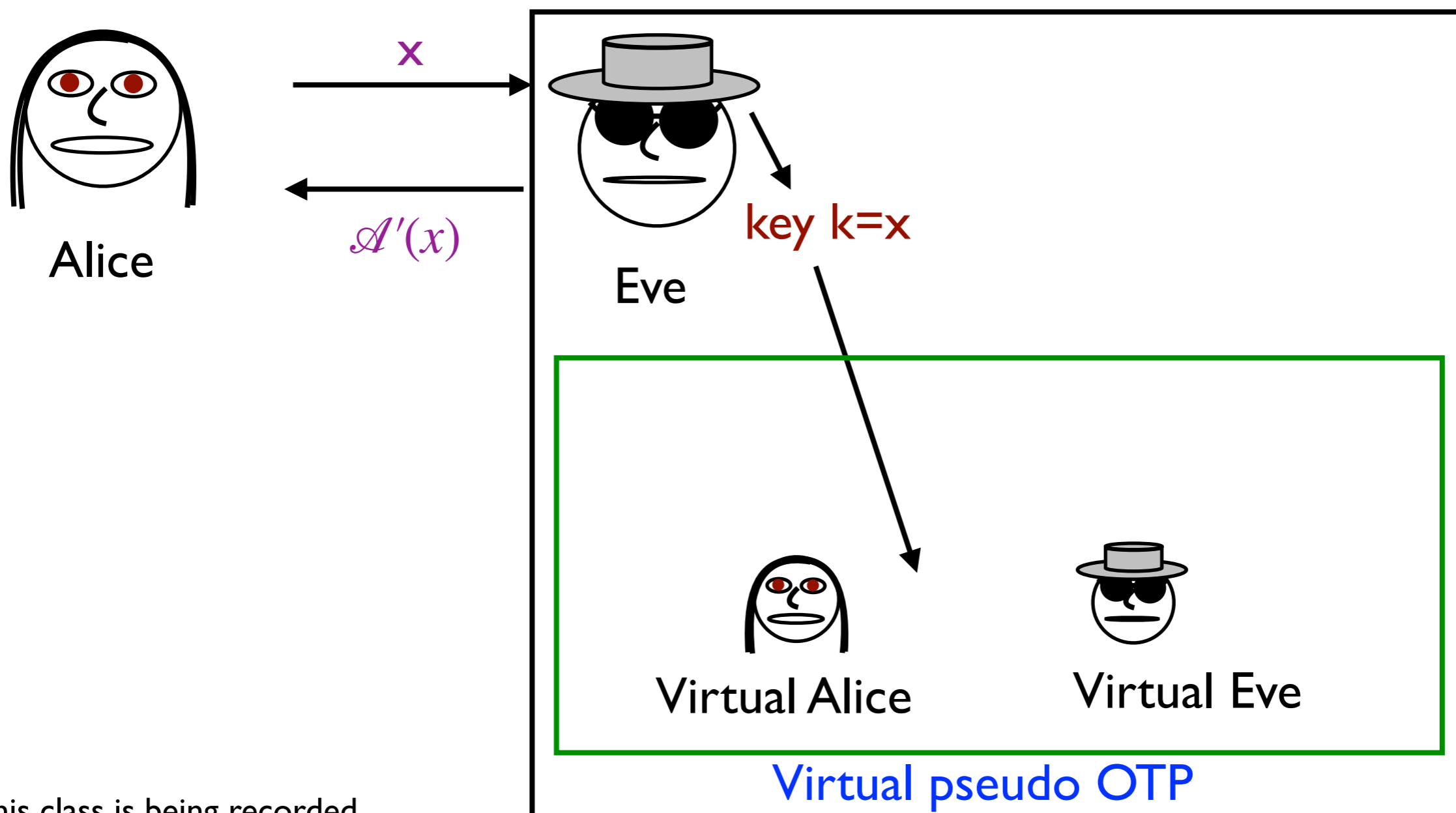
Given an attack $(\mathcal{B}(s), \mathcal{A}(c))$ on the pseudo one-time pad, we will construct an attack $\mathcal{A}'(x)$ on the pseudorandom generator.

$\mathcal{A}'(x)$ will succeed in distinguishing the pseudorandom numbers from random with probability $\epsilon'(s)$ if $(\mathcal{B}(s), \mathcal{A}(c))$ succeeds in distinguishing messages with probability $\epsilon(s)$. If the pseudorandom generator is secure, $\epsilon'(s)$ will be negligible, which will imply that $\epsilon(s)$ is also negligible. Therefore this particular attack on the pseudo one-time pad doesn't succeed.

But the attack was arbitrary, so the pseudo one-time pad is EAV-secure.

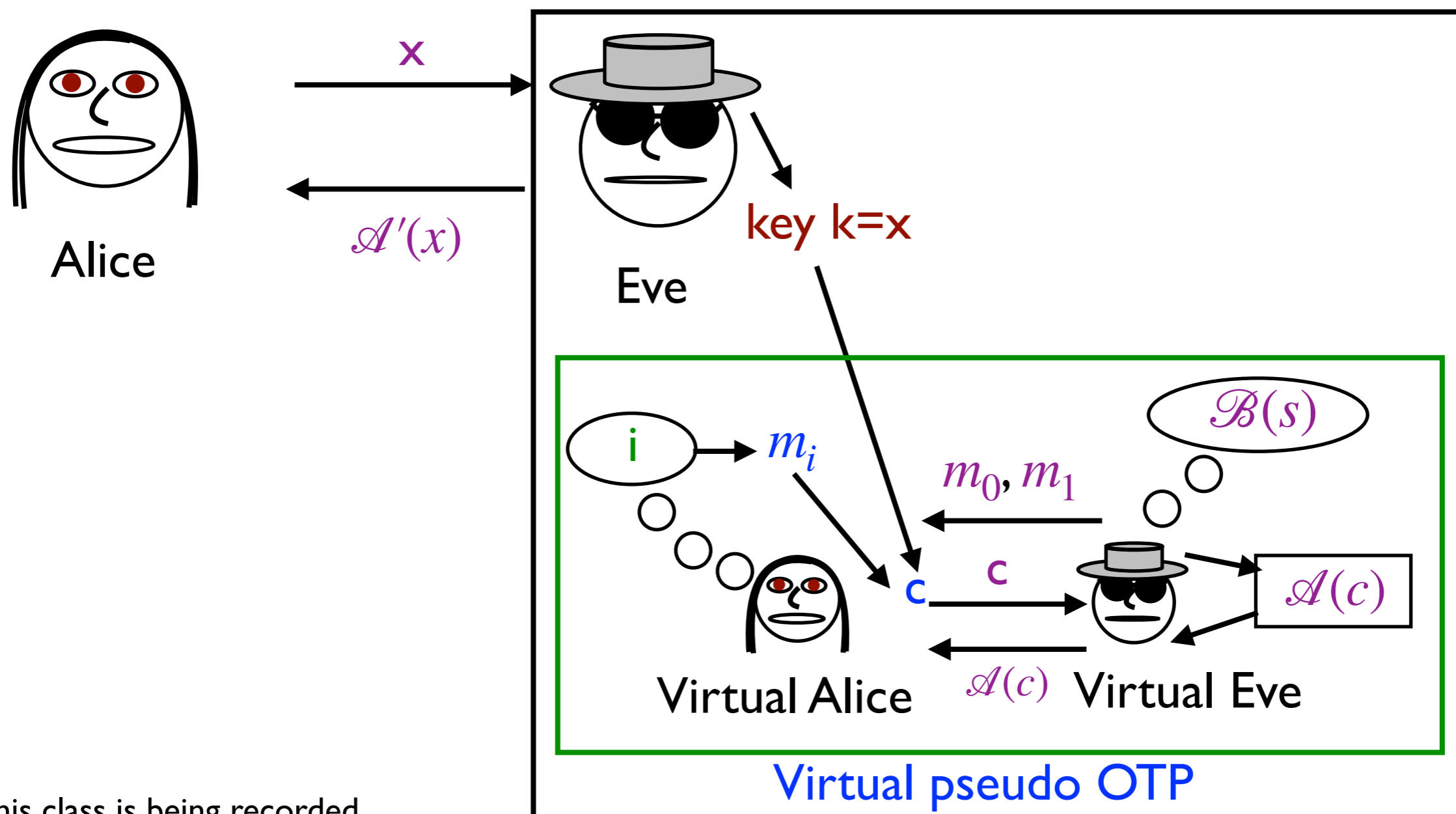
The Reduction

The idea is that the attack $\mathcal{A}'(x)$ will use x as the key in a virtual pseudo one-time pad protocol, in which a virtual Eve will run the attack $(\mathcal{B}(s), \mathcal{A}(c))$ on the pseudo one-time pad.



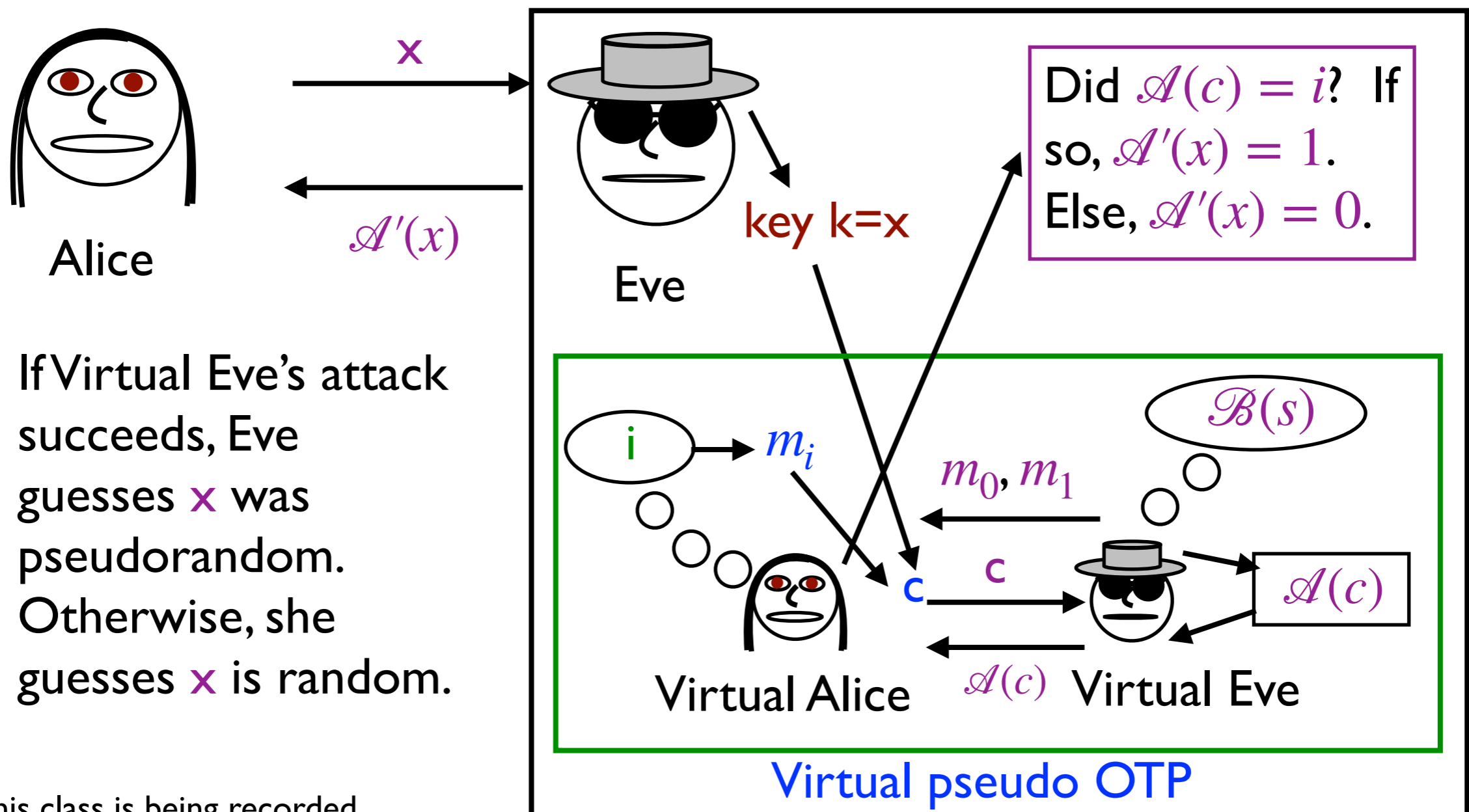
The Reduction

The idea is that the attack $\mathcal{A}'(x)$ will use x as the key in a virtual pseudo one-time pad protocol, in which a virtual Eve will run the attack $(\mathcal{B}(s), \mathcal{A}(c))$ on the pseudo one-time pad.



The Reduction

The idea is that the attack $\mathcal{A}'(x)$ will use x as the key in a virtual pseudo one-time pad protocol, in which a virtual Eve will run the attack $(\mathcal{B}(s), \mathcal{A}(c))$ on the pseudo one-time pad.



Success Probability for Random x

In the case where x was actually a **uniform random string** chosen by Alice, then the virtual protocol run was actually a virtual one-time pad. This means that it has perfect secrecy, and whatever pair of messages (m_0, m_1) was chosen by $\mathcal{B}(s)$ and whichever ciphertext c ended up being used,

$$\Pr(\text{Enc}(x, m_0) = c) = \Pr(\text{Enc}(x, m_1) = c) = p_c$$

Now,

$$\begin{aligned}\Pr(\mathcal{A}(\text{Enc}(x, m_i)) = 1) &= \sum_c \Pr(\text{Enc}(x, m_i) = c) \Pr(\mathcal{A}(c) = 1) \\ &= \sum_c p_c \Pr(\mathcal{A}(c) = 1)\end{aligned}$$

which doesn't depend on i . In particular,

$$\Pr(\mathcal{A}(\text{Enc}(x, m_0)) = 1) = \Pr(\mathcal{A}(\text{Enc}(x, m_1)) = 1)$$

Success Probability for $\mathcal{A}'(x)$ (random)

To calculate the success probability for $\mathcal{A}'(x)$, first note that

$$\Pr(\mathcal{A}'(x) = 1) = \frac{1}{2} [\Pr(\mathcal{A}(\text{Enc}(x, m_1)) = 1) + 1 - \Pr(\mathcal{A}(\text{Enc}(x, m_0)) = 1)]$$

(Actually this should be averaged over pairs (m_0, m_1) chosen by $\mathcal{B}(s)$, but we can specialize to $\mathcal{B}(s)$ with deterministic output.)

If x is random, then $\Pr(\mathcal{A}'(x) = 1) = 1/2$.

$\mathcal{A}'(G(y))$ (pseudorandom case)

In the case that $x = G(y)$ is pseudorandom, then let

$$\delta(x) = \Pr(\mathcal{A}(\text{Enc}(G(y), m_1)) = 1) - \Pr(\mathcal{A}(\text{Enc}(G(y), m_0)) = 1)$$

so $\Pr(\mathcal{A}'(G(y)) = 1) = 1/2 (1 + \delta(G(y)))$ in this case. Note that here, probability is taken over randomness of Enc and \mathcal{A} , but we are not yet averaging over y .

When we do average over possible x and y values,

$$\Pr_y(\mathcal{A}'(G(y)) = 1) - \Pr_x(\mathcal{A}'(x) = 1) = \frac{1}{2} \sum_y \Pr(y) \delta(G(y))$$

so by the definition of a pseudorandom generator,

$$\left| \sum_y \Pr(y) \delta(G(y)) \right| \leq 2\epsilon(s)$$

for negligible $\epsilon(s)$, which is also the formula needed for EAV security of the pseudo one-time pad.

Complexity of Reduction

One additional thing that needs to be checked is the complexity of the attack $\mathcal{A}'(x)$.

- Running the Virtual Alice in the one-time pad simulation takes a time linear in the message length $O(\ell(s))$.
- Running Virtual Eve takes the time required to run the attack $(\mathcal{B}(s), \mathcal{A}(c))$, which is polynomial in the message length.
- Translating the outcome of the simulated one-time pad into the attack $\mathcal{A}'(x)$ takes constant time.

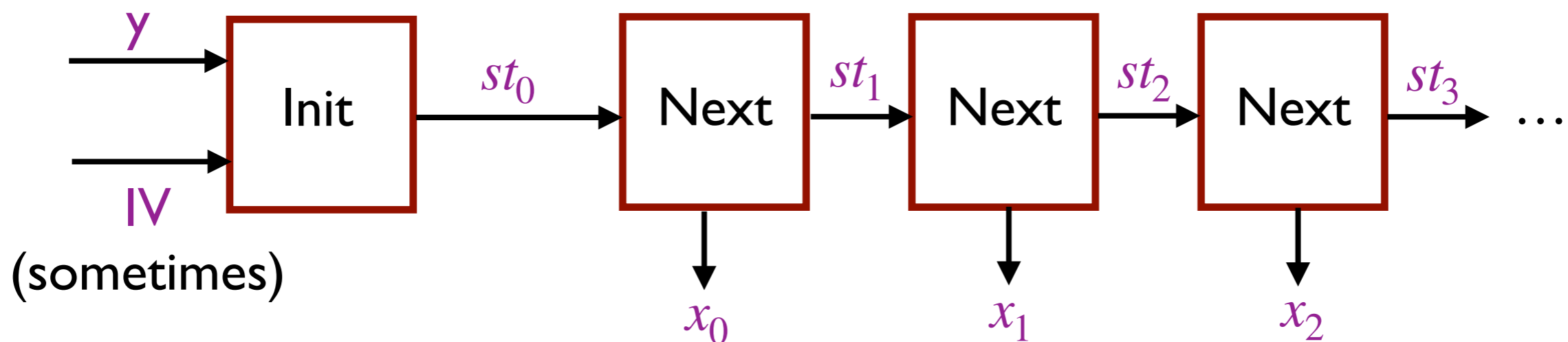
Therefore the whole attack takes time polynomial in $\ell(s)$, which is as it should be.

Note: “Polynomial time” mean polynomial in the size of the input to the algorithm, which is here $\ell(s)$, not necessarily polynomial in s .

Stream Ciphers

Pseudorandom generators are a lot more efficient than using truly random bits, but are still a bit clunky in that they can only output big chunks of bits. You need to know how many bits $\ell(s)$ you are going to need when you pick s (and therefore when you establish your key if you are doing the pseudo one-time pad).

Stream ciphers are a solution: given a seed, they generate a stream of pseudorandom bits as you desire.

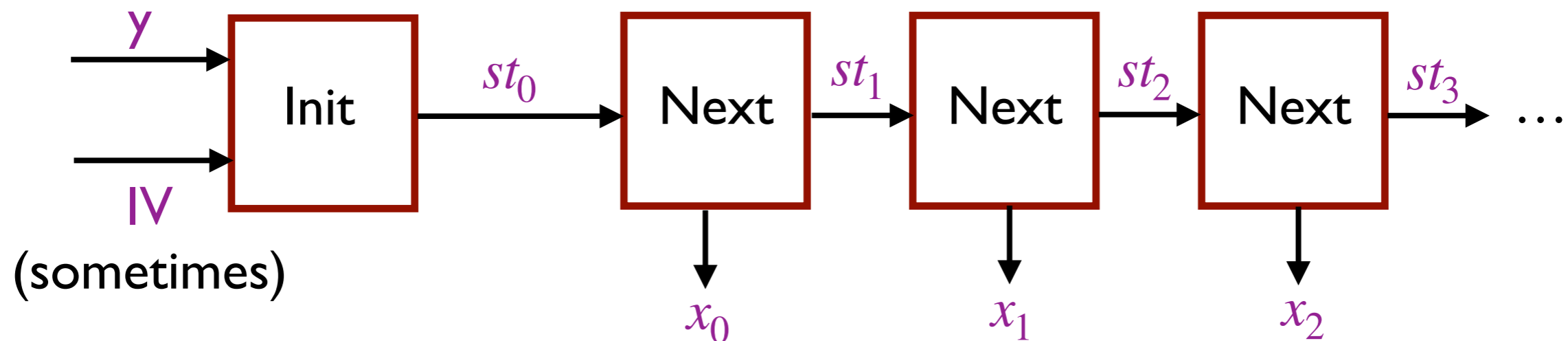


Stream Cipher Definition

A **stream cipher** is a pair of *deterministic efficient* algorithms (**Init**, **Next**).

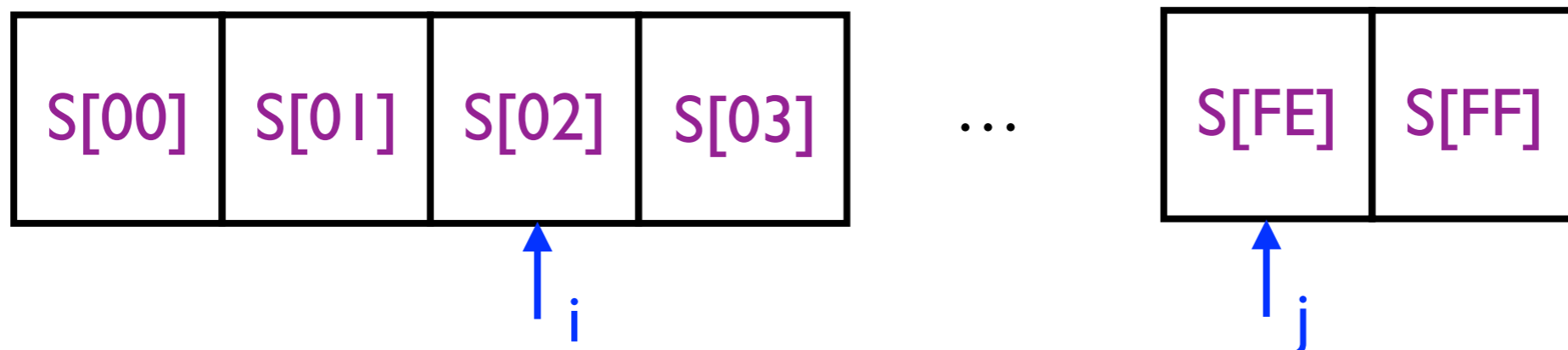
- **Init** is the initialization algorithm; it takes a seed y as input and sometimes an **initial value IV** and outputs an initial state st_0 .
- **Next** is the algorithm that produces bits; it takes as input a current state st_i and outputs a bit x_i and a new state st_{i+1} .

When there is no IV, the stream cipher is **secure** if the function $G_n(y) = x_0x_1x_2\dots x_n$ is a pseudorandom generator for any n which is polynomial in $|y| = s$.



RC4 Overview

Invented by Ron Rivest, RC4 was widely used for many years, for instance as part of the first wi-fi protocol, WEP.



RC4 is a stream cipher. The state passed between steps consists of a permutation of the numbers 0...255 (realized as an array of 256 bytes, each containing a distinct number), plus two additional bytes *i* and *j*, which are pointers into the array.

It works by performing swaps between locations in the array in a somewhat complicated way. One entry is then returned as the next output in the stream.

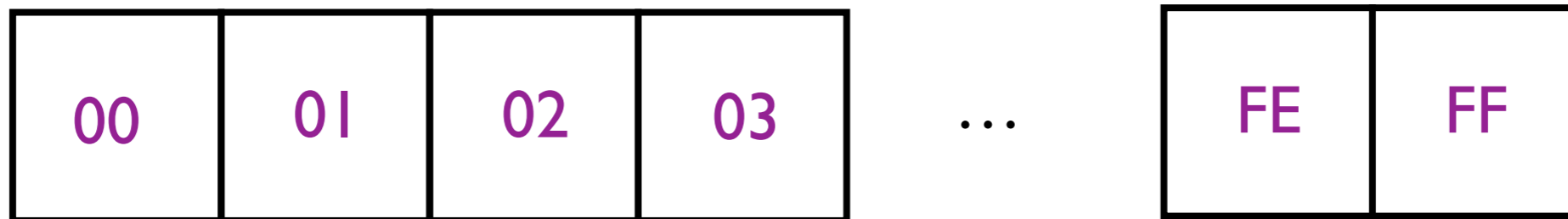
RC4 Init



$$k = (03, FA, \dots)$$

RC4 takes no IV, and the key k can have a variable length s up to 255 bytes. Let k_i be the i th byte of k if $i < s$, or the $(i \bmod s)$ th byte of k when $i \geq s$.

RC4 Init

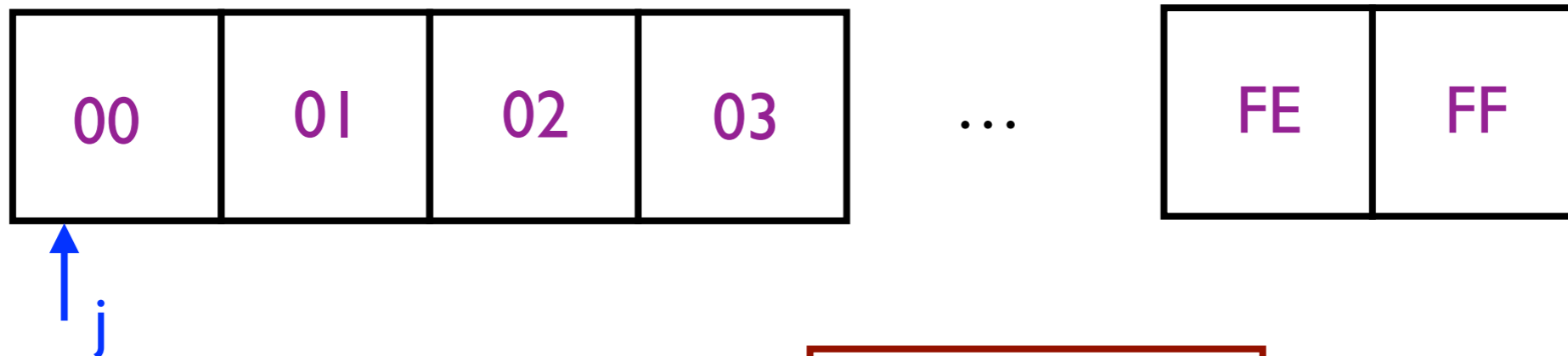


$$k = (03, FA, \dots)$$

RC4 takes no IV, and the key k can have a variable length s up to 255 bytes. Let k_i be the i th byte of k if $i < s$, or the $(i \bmod s)$ th byte of k when $i \geq s$.

- First, initialize $S[i] := i$ for all $i = 0, \dots, 255$.

RC4 Init

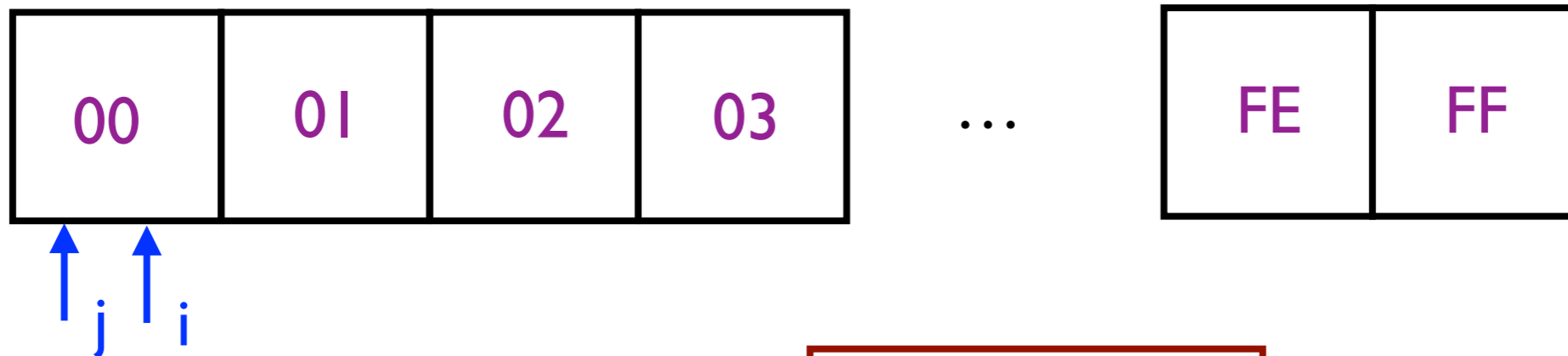


$$k = (03, FA, \dots)$$

RC4 takes no IV, and the key k can have a variable length s up to 255 bytes. Let k_i be the i th byte of k if $i < s$, or the $(i \bmod s)$ th byte of k when $i \geq s$.

- First, initialize $S[i] := i$ for all $a=0, \dots, 255$.
- Start with $j := 0$.

RC4 Init

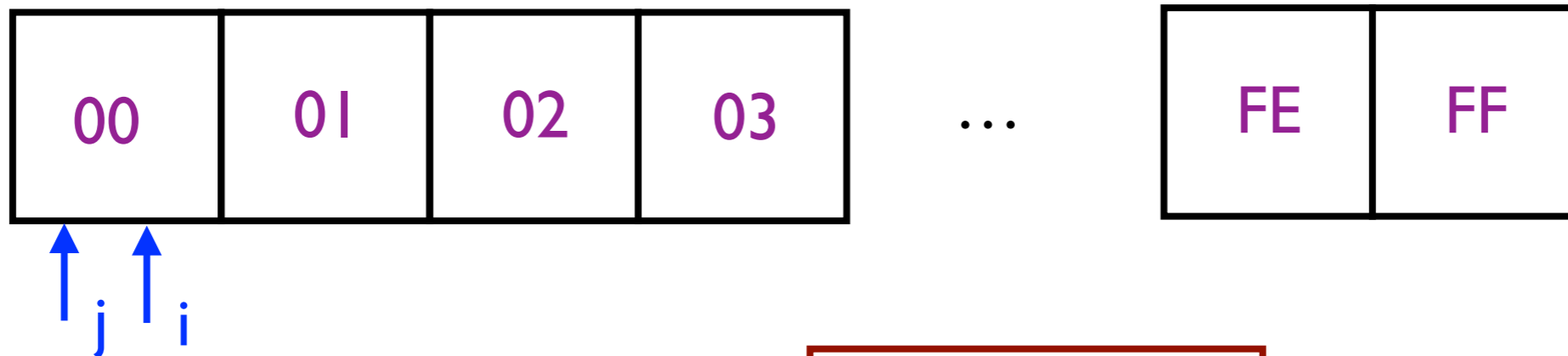


$$k = (03, FA, \dots)$$

RC4 takes no IV, and the key k can have a variable length s up to 255 bytes. Let k_i be the i th byte of k if $i < s$, or the $(i \bmod s)$ th byte of k when $i \geq s$.

- First, initialize $S[i] := i$ for all $a=0, \dots, 255$.
- Start with $j := 0$.
- For $i = 0$ to 255:

RC4 Init

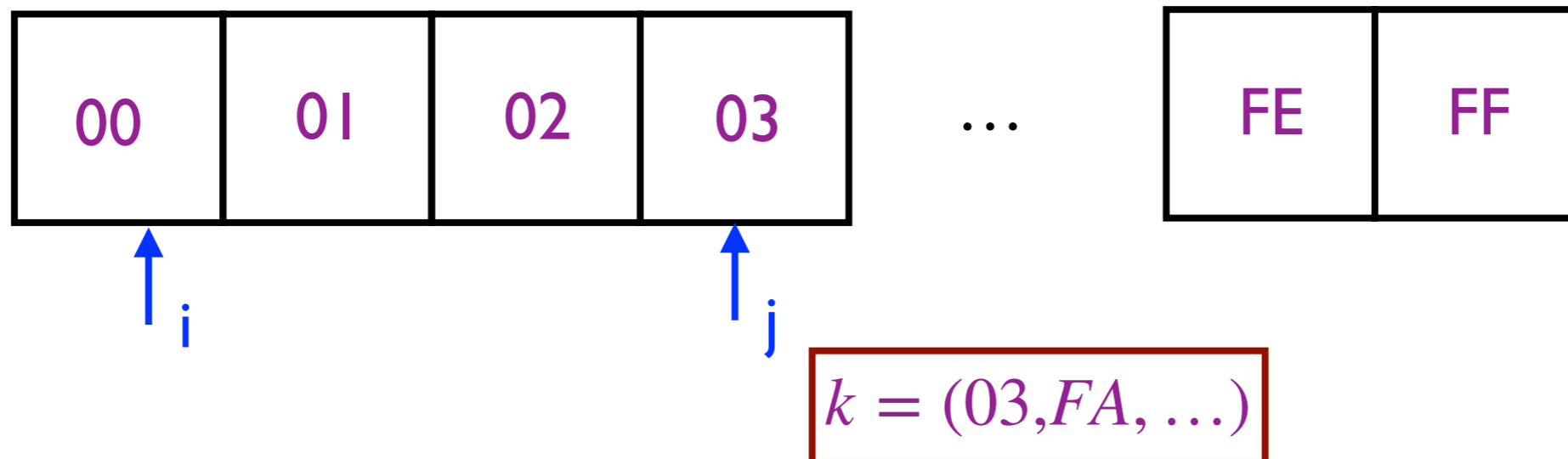


$$k = (03, FA, \dots)$$

RC4 takes no IV, and the key k can have a variable length s up to 255 bytes. Let k_i be the i th byte of k if $i < s$, or the $(i \bmod s)$ th byte of k when $i \geq s$.

- First, initialize $S[i] := i$ for all $a=0, \dots, 255$.
- Start with $j := 0$.
- For $i = 0$ to 255:
 - ▶ $j := j + S[i] + k_i \bmod 256$

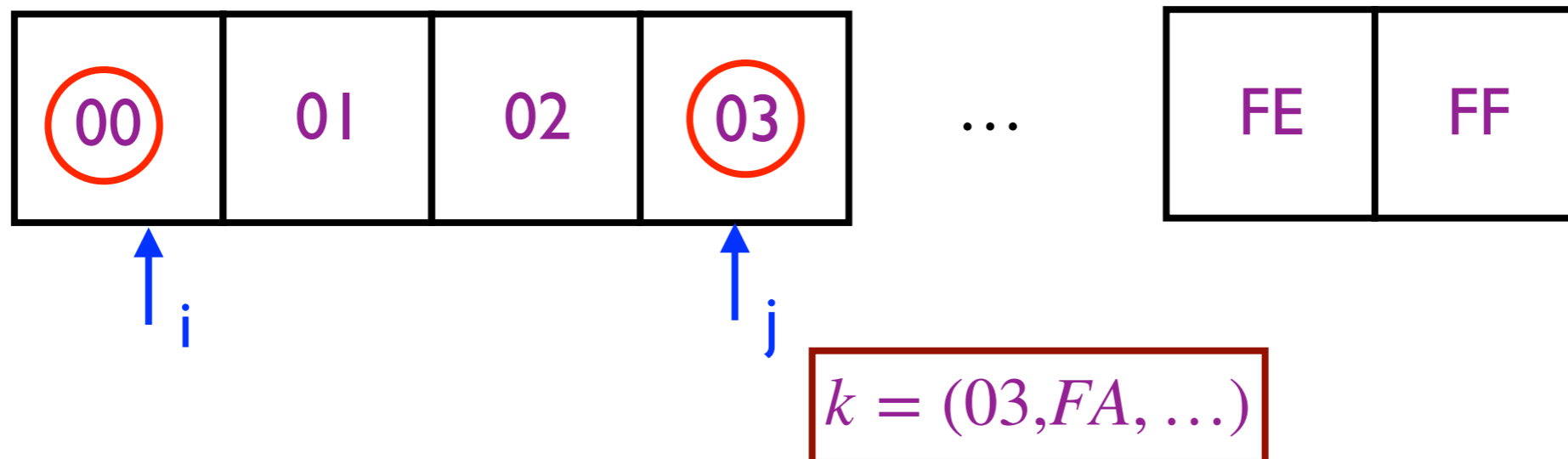
RC4 Init



RC4 takes no IV, and the key k can have a variable length s up to 255 bytes. Let k_i be the i th byte of k if $i < s$, or the $(i \bmod s)$ th byte of k when $i \geq s$.

- First, initialize $S[i] := i$ for all $a=0, \dots, 255$.
- Start with $j := 0$.
- For $i = 0$ to 255:
 - ▶ $j := j + S[i] + k_i \bmod 256$

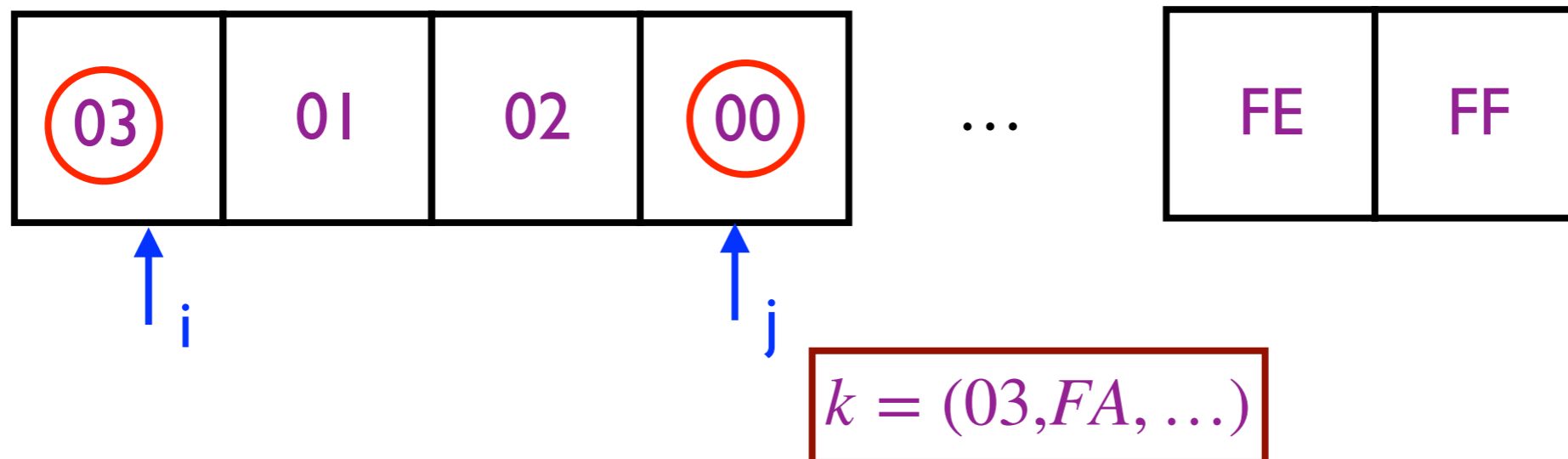
RC4 Init



RC4 takes no IV, and the key k can have a variable length s up to 255 bytes. Let k_i be the i th byte of k if $i < s$, or the $(i \bmod s)$ th byte of k when $i \geq s$.

- First, initialize $S[i] := i$ for all $a=0, \dots, 255$.
- Start with $j := 0$.
- For $i = 0$ to 255:
 - ▶ $j := j + S[i] + k_i \bmod 256$
 - ▶ Swap $S[i]$ and $S[j]$.

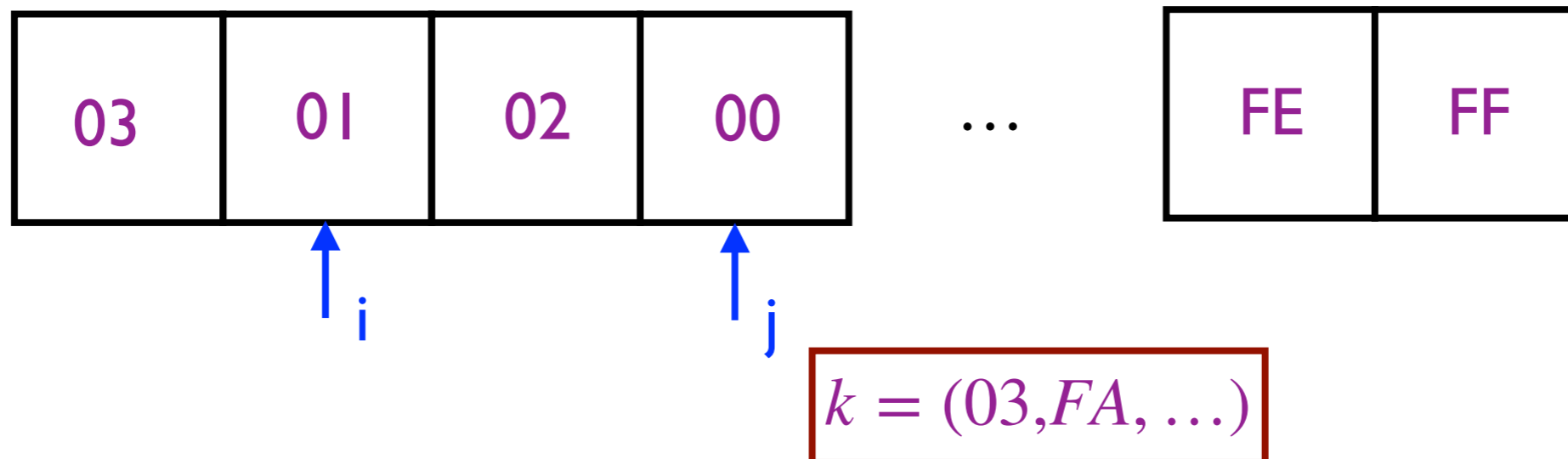
RC4 Init



RC4 takes no IV, and the key k can have a variable length s up to 255 bytes. Let k_i be the i th byte of k if $i < s$, or the $(i \bmod s)$ th byte of k when $i \geq s$.

- First, initialize $S[i] := i$ for all $a=0, \dots, 255$.
- Start with $j := 0$.
- For $i = 0$ to 255 :
 - ▶ $j := j + S[i] + k_i \bmod 256$
 - ▶ Swap $S[i]$ and $S[j]$.

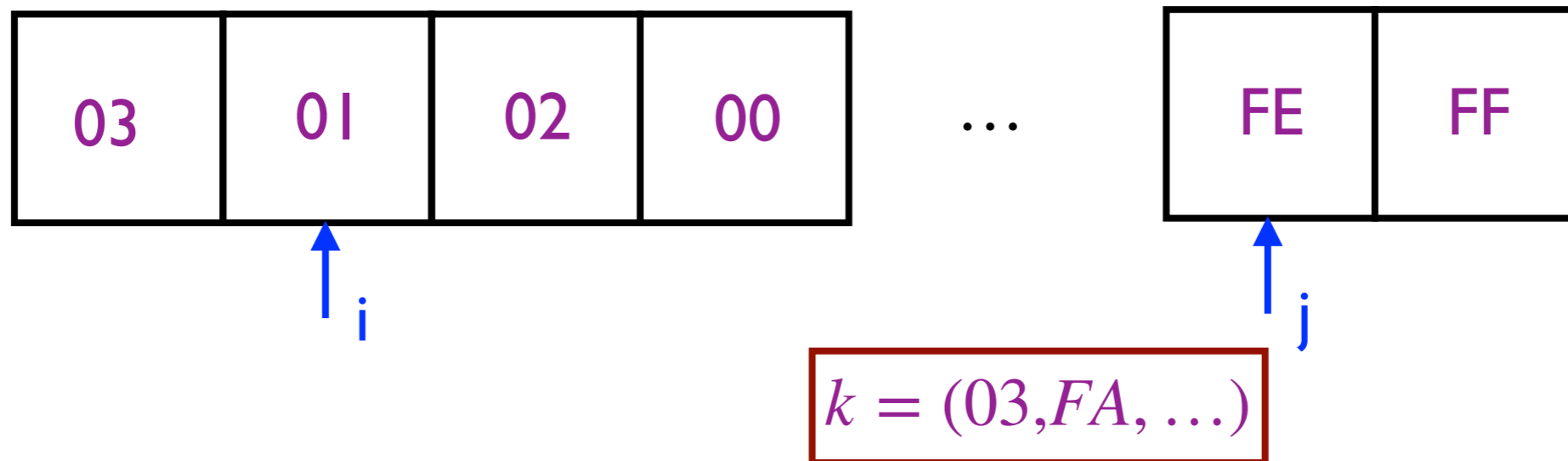
RC4 Init



RC4 takes no IV, and the key k can have a variable length s up to 255 bytes. Let k_i be the i th byte of k if $i < s$, or the $(i \bmod s)$ th byte of k when $i \geq s$.

- First, initialize $S[i] := i$ for all $a=0, \dots, 255$.
- Start with $j := 0$.
- For $i = 0$ to 255:
 - ▶ $j := j + S[i] + k_i \bmod 256$
 - ▶ Swap $S[i]$ and $S[j]$.

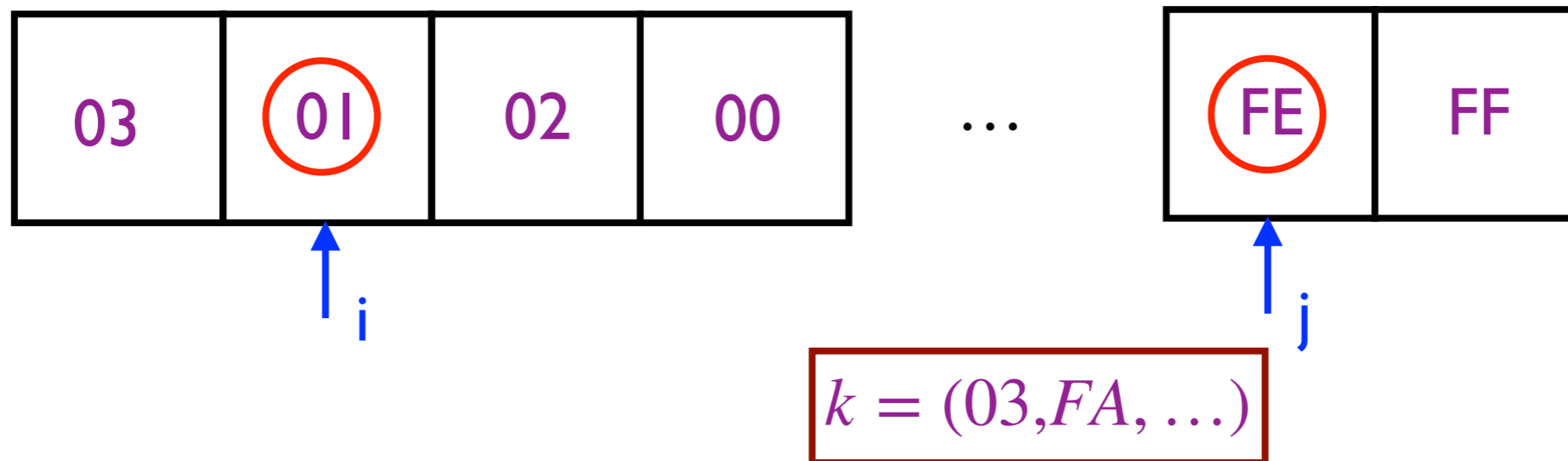
RC4 Init



RC4 takes no IV, and the key k can have a variable length s up to 255 bytes. Let k_i be the i th byte of k if $i < s$, or the $(i \bmod s)$ th byte of k when $i \geq s$.

- First, initialize $S[i] := i$ for all $a=0, \dots, 255$.
- Start with $j := 0$.
- For $i = 0$ to 255:
 - ▶ $j := j + S[i] + k_i \bmod 256$
 - ▶ Swap $S[i]$ and $S[j]$.

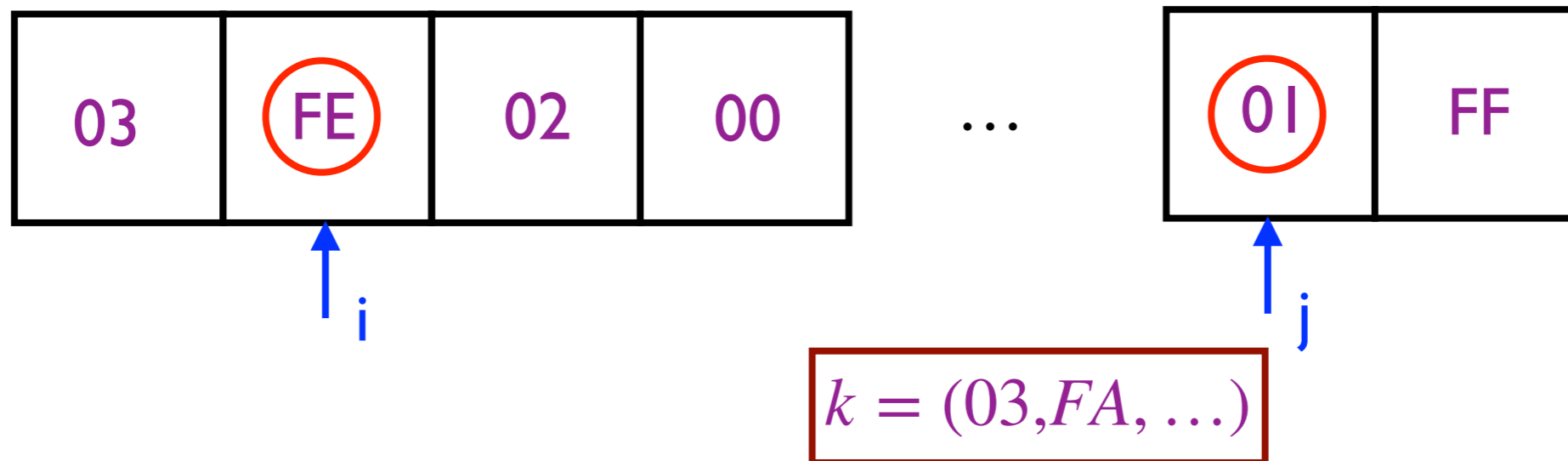
RC4 Init



RC4 takes no IV, and the key k can have a variable length s up to 255 bytes. Let k_i be the i th byte of k if $i < s$, or the $(i \bmod s)$ th byte of k when $i \geq s$.

- First, initialize $S[i] := i$ for all $a=0, \dots, 255$.
- Start with $j := 0$.
- For $i = 0$ to 255:
 - ▶ $j := j + S[i] + k_i \bmod 256$
 - ▶ Swap $S[i]$ and $S[j]$.

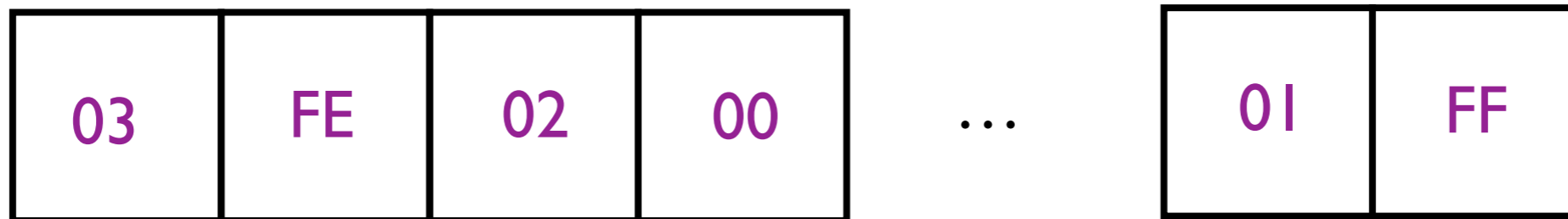
RC4 Init



RC4 takes no IV, and the key k can have a variable length s up to 255 bytes. Let k_i be the i th byte of k if $i < s$, or the $(i \bmod s)$ th byte of k when $i \geq s$.

- First, initialize $S[i] := i$ for all $a=0, \dots, 255$.
- Start with $j := 0$.
- For $i = 0$ to 255:
 - ▶ $j := j + S[i] + k_i \bmod 256$
 - ▶ Swap $S[i]$ and $S[j]$.

RC4 Init

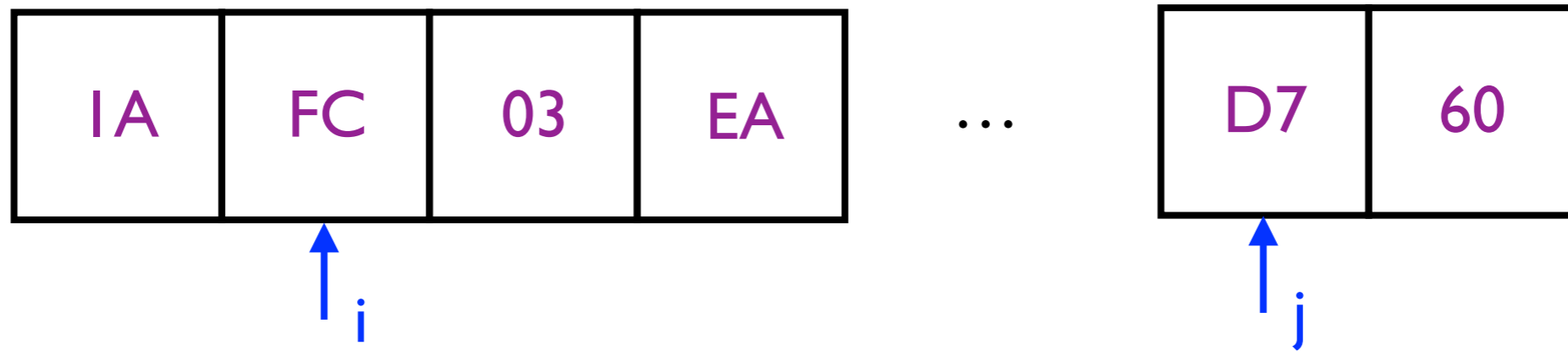


$$k = (03, FA, \dots)$$

RC4 takes no IV, and the key k can have a variable length s up to 255 bytes. Let k_i be the i th byte of k if $i < s$, or the $(i \bmod s)$ th byte of k when $i \geq s$.

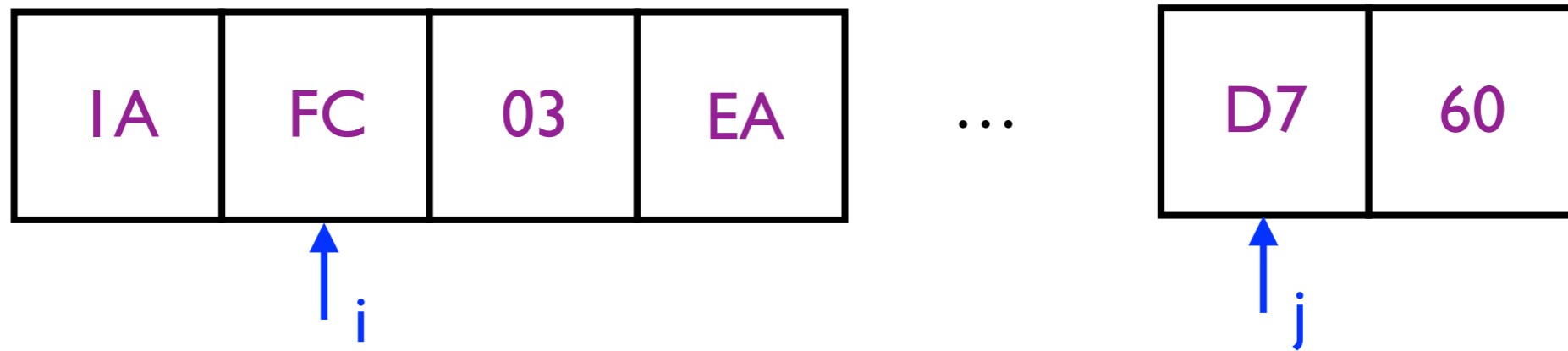
- First, initialize $S[i] := i$ for all $a=0, \dots, 255$.
- Start with $j := 0$.
- For $i = 0$ to 255:
 - ▶ $j := j + S[i] + k_i \bmod 256$
 - ▶ Swap $S[i]$ and $S[j]$.
- Reset $i := 0, j := 0$.

RC4 Next



The **Next** algorithm has the following steps:

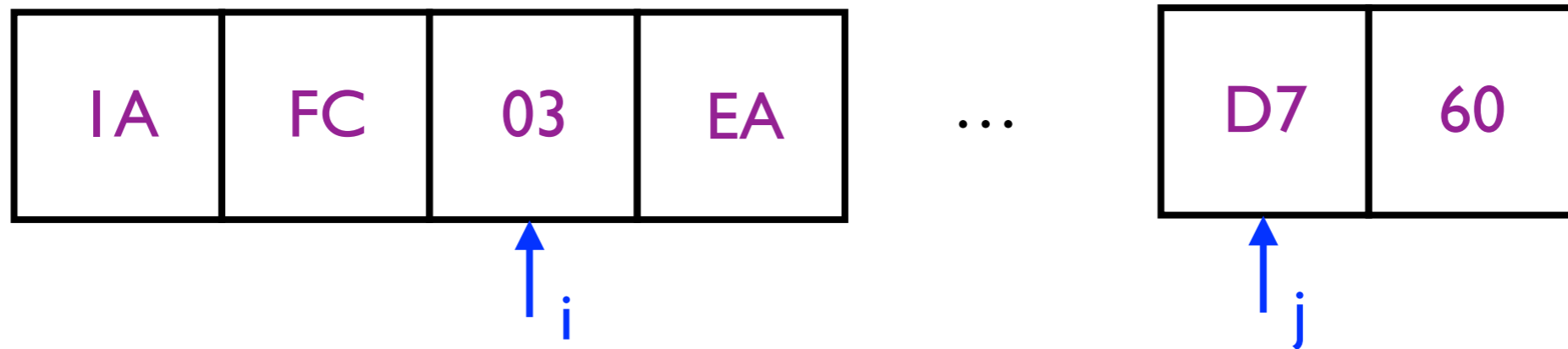
RC4 Next



The **Next** algorithm has the following steps:

- $i := i + 1$

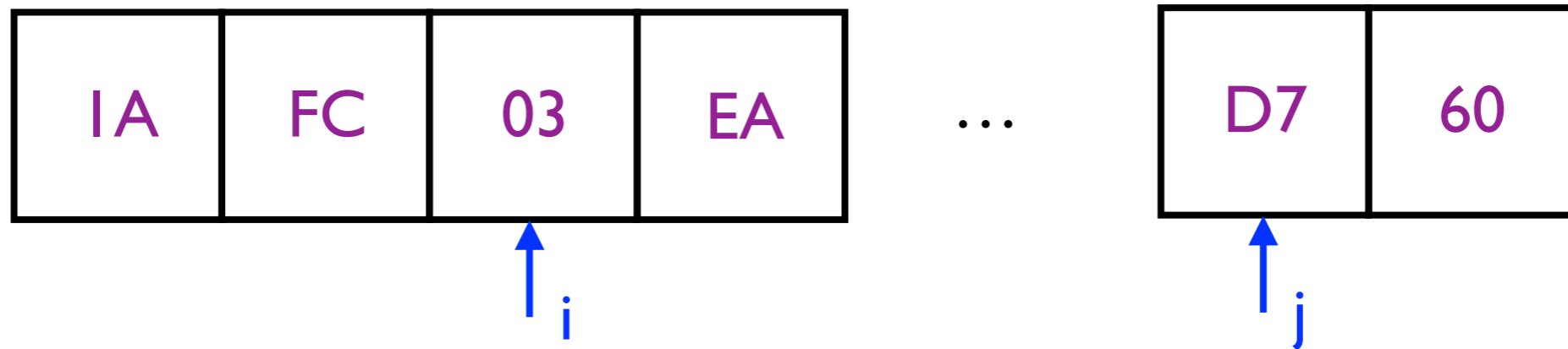
RC4 Next



The **Next** algorithm has the following steps:

- $i := i + 1$

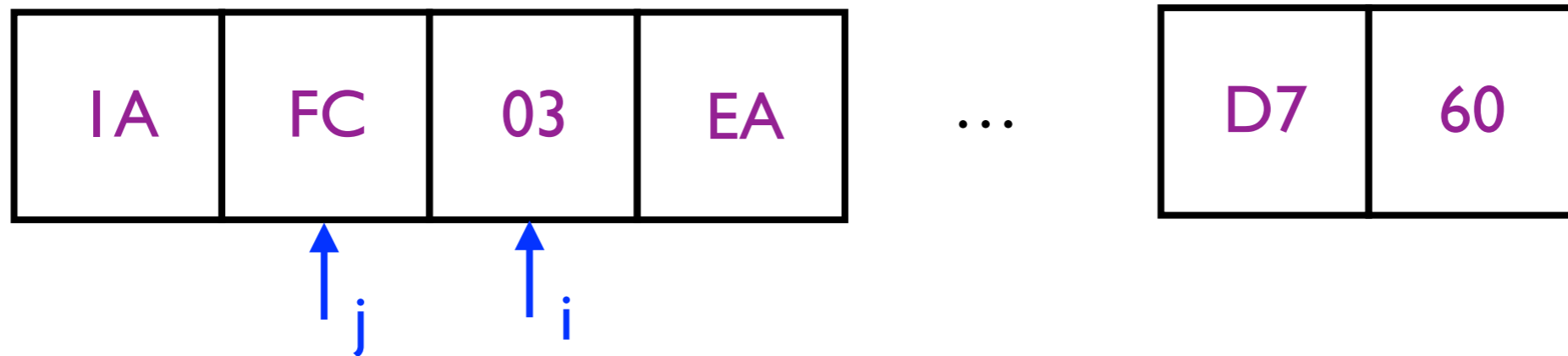
RC4 Next



The **Next** algorithm has the following steps:

- $i := i + 1$
- $j := j + S[i]$

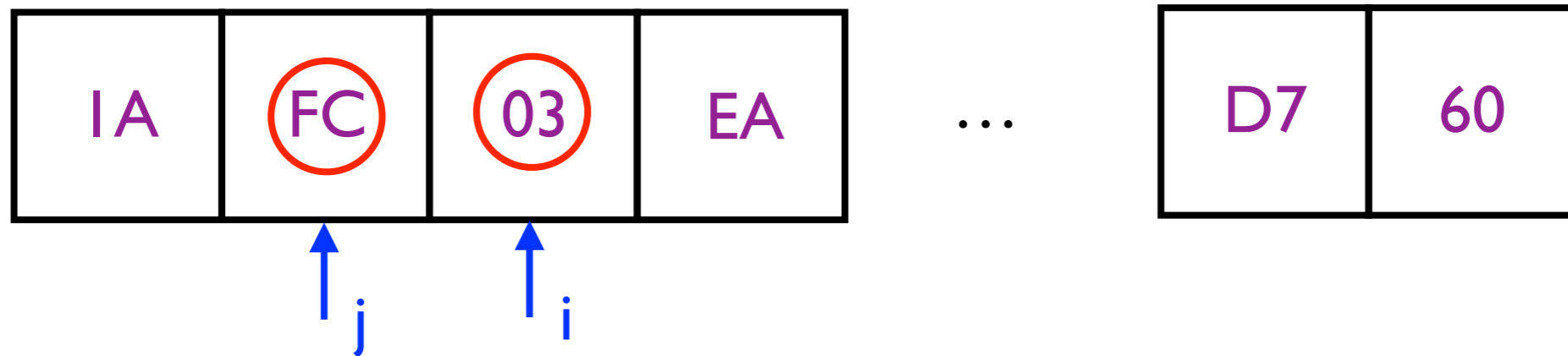
RC4 Next



The **Next** algorithm has the following steps:

- $i := i + 1$
- $j := j + S[i]$

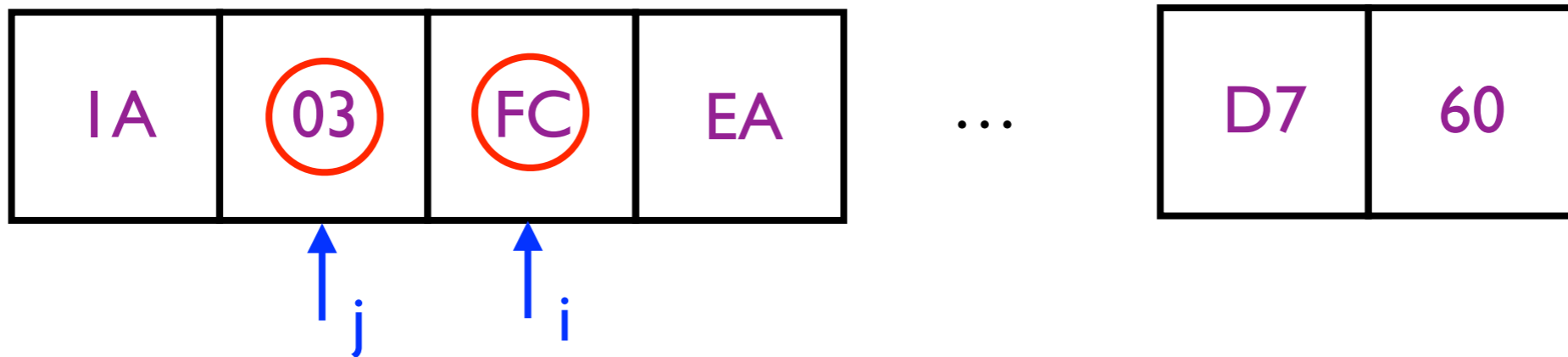
RC4 Next



The **Next** algorithm has the following steps:

- $i := i + 1$
- $j := j + S[i]$
- Swap $S[i]$ and $S[j]$

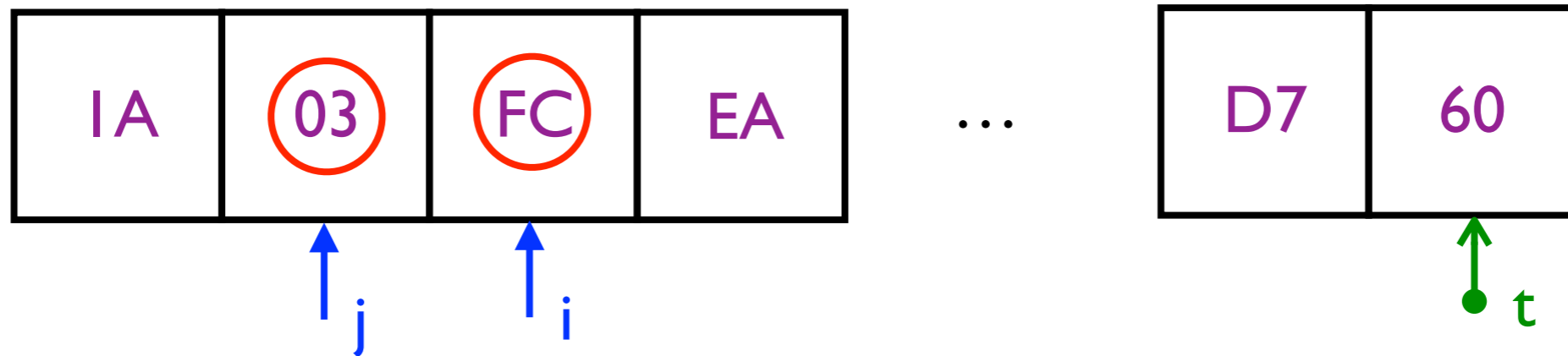
RC4 Next



The **Next** algorithm has the following steps:

- $i := i + 1$
- $j := j + S[i]$
- Swap $S[i]$ and $S[j]$

RC4 Next



The **Next** algorithm has the following steps:

- $i := i + 1$
- $j := j + S[i]$
- Swap $S[i]$ and $S[j]$
- $t := S[i] + S[j]$

RC4 Next



The **Next** algorithm has the following steps:

- $i := i + 1$
- $j := j + S[i]$
- Swap $S[i]$ and $S[j]$
- $t := S[i] + S[j]$
- $y := S[t]$

RC4 Next



The **Next** algorithm has the following steps:

- $i := i + 1$
- $j := j + S[i]$
- Swap $S[i]$ and $S[j]$
- $t := S[i] + S[j]$
- $y := S[t]$

Then y is the output byte and the state is passed to the following **Next** call.

The idea here is that if the entries $S[i]$ are essentially random, then t is random too and y is the output of a location with no clear relation to i and j .

RC4 Security

Because it relies only on swaps and simple arithmetic, RC4 has simple fast implementations. But is it secure?

Vote: Can we prove RC4 is secure? (Yes/No)

RC4 Security

Because it relies only on swaps and simple arithmetic, RC4 has simple fast implementations. But is it secure?

Vote: Can we prove RC4 is secure? (Yes/No)

No, it would be hard to prove security because of how complicated it is, and we wouldn't in any case be able to prove security more than conditionally. But more importantly, this is a **fixed-size** protocol. The limit on key size is 256 bytes, so the time to attack it is a constant, at most about 256^{256} .

Instead security here means that, in practice, there is no known attack that is much better than brute force.

RC4 Security

Because it relies only on swaps and simple arithmetic, RC4 has simple fast implementations. But is it secure?

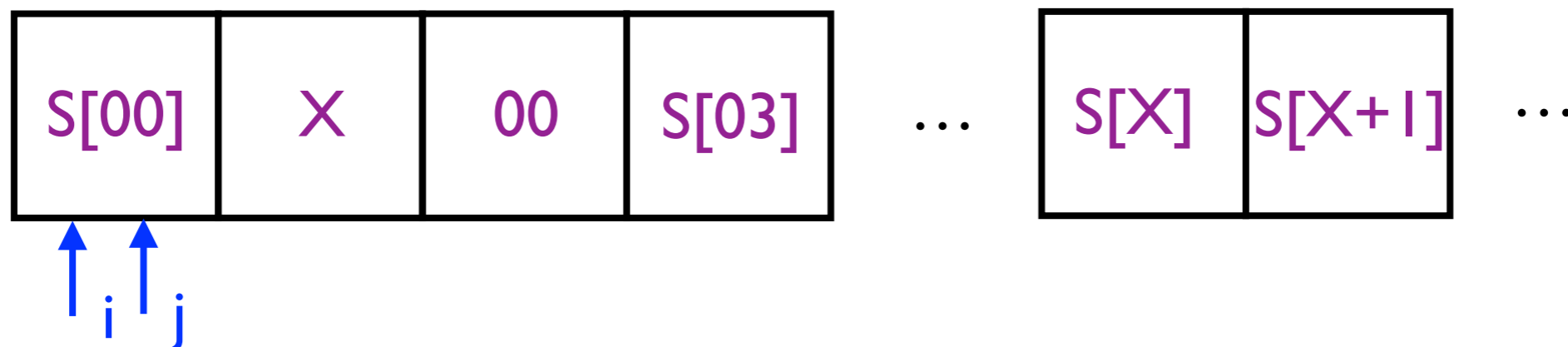
Vote: Can we prove RC4 is secure? (Yes/No)

No, it would be hard to prove security because of how complicated it is, and we wouldn't in any case be able to prove security more than conditionally. But more importantly, this is a **fixed-size** protocol. The limit on key size is 256 bytes, so the time to attack it is a constant, at most about 256^{256} .

Instead security here means that, in practice, there is no known attack that is much better than brute force.

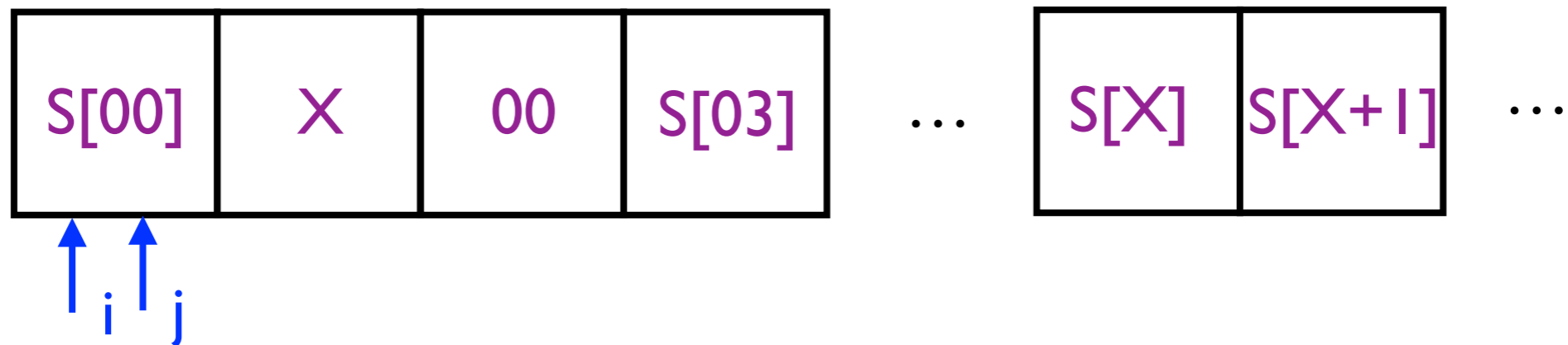
Vote: Is RC4 secure in this sense? (Yes/No)

An Attack on RC4



Suppose that after the `Init` algorithm, we can imagine the state of the array to be essentially a random permutation. With probability about $1/256$, $S[02] = 00$ and $S[01] = X \neq 02$. In this case,

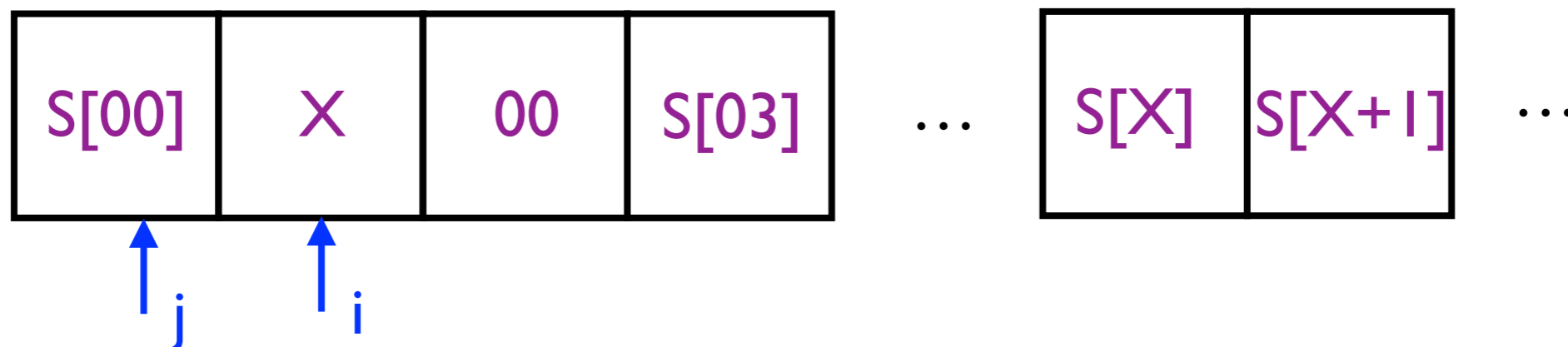
An Attack on RC4



Suppose that after the `Init` algorithm, we can imagine the state of the array to be essentially a random permutation. With probability about $1/256$, $S[02] = 00$ and $S[01] = X \neq 02$. In this case,

- The 1st step sends i to 01 , j to X , and swaps $S[01]$ and $S[X]$.

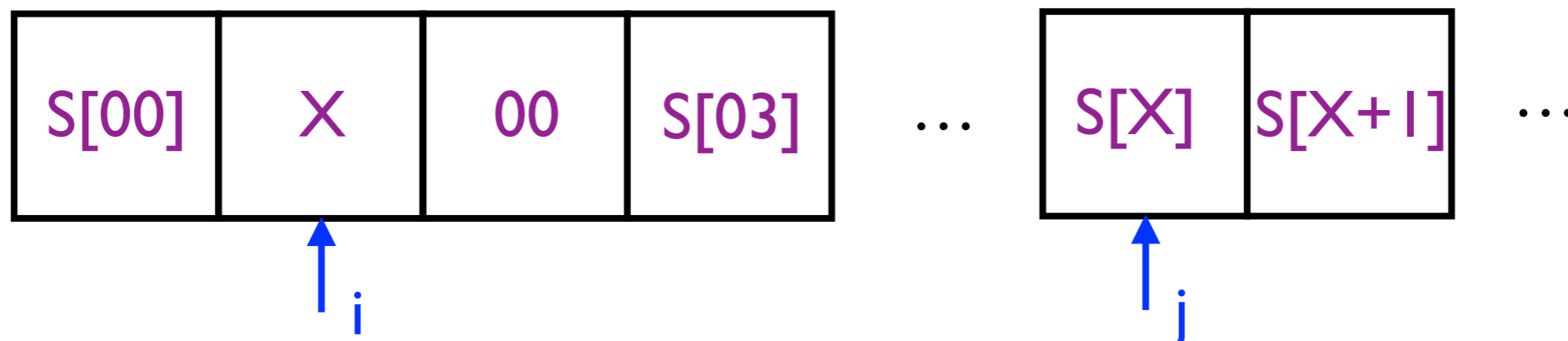
An Attack on RC4



Suppose that after the `Init` algorithm, we can imagine the state of the array to be essentially a random permutation. With probability about $1/256$, $S[02] = 00$ and $S[01] = X \neq 02$. In this case,

- The 1st step sends i to 01 , j to X , and swaps $S[01]$ and $S[X]$.

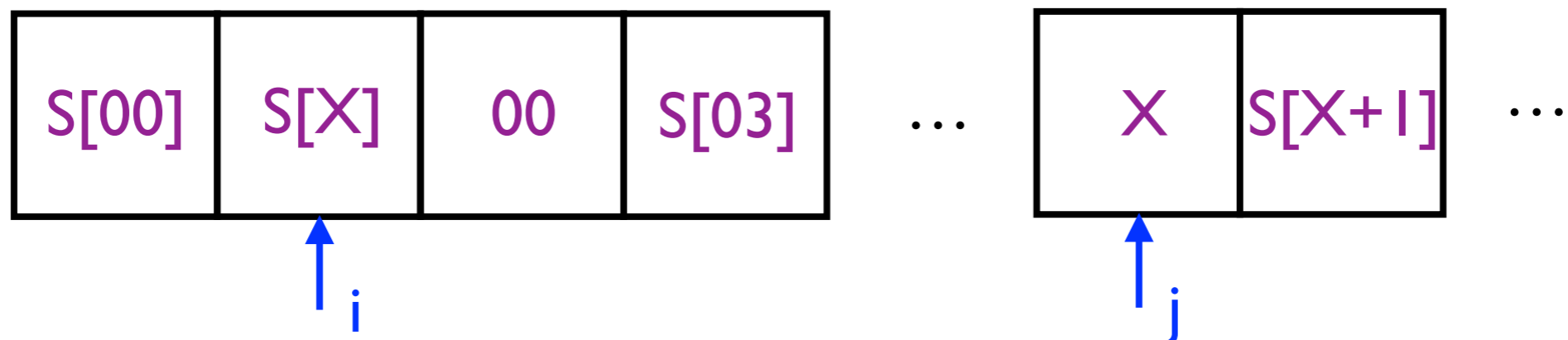
An Attack on RC4



Suppose that after the `Init` algorithm, we can imagine the state of the array to be essentially a random permutation. With probability about $1/256$, $S[02] = 00$ and $S[01] = X \neq 02$. In this case,

- The 1st step sends i to 01 , j to X , and swaps $S[01]$ and $S[X]$.

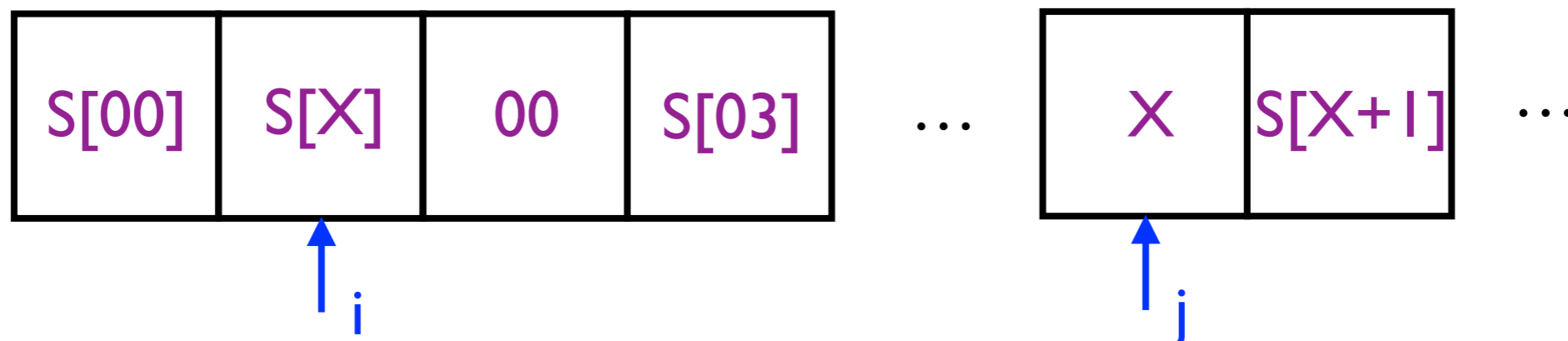
An Attack on RC4



Suppose that after the **Init** algorithm, we can imagine the state of the array to be essentially a random permutation. With probability about $1/256$, $S[02] = 00$ and $S[01] = X \neq 02$. In this case,

- The 1st step sends i to 01 , j to X , and swaps $S[01]$ and $S[X]$.

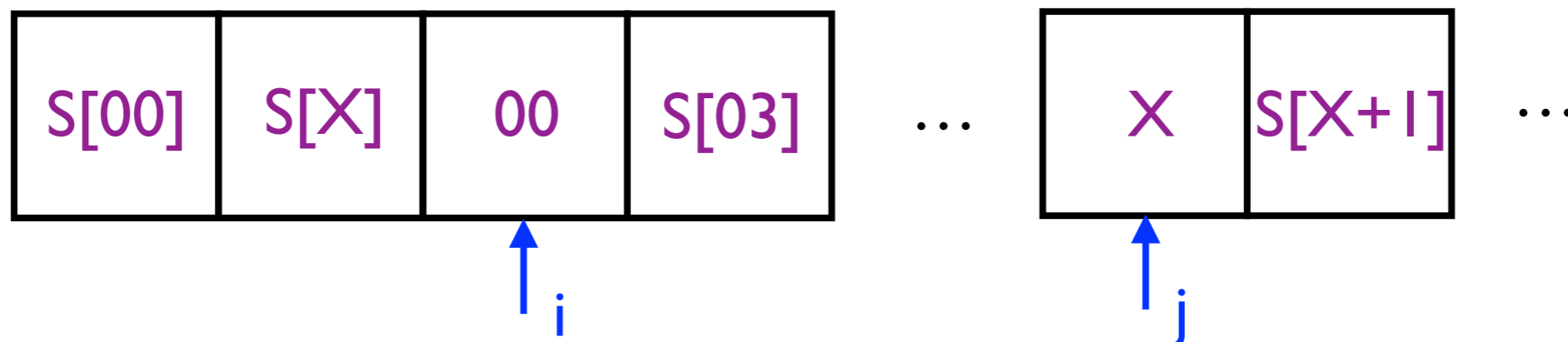
An Attack on RC4



Suppose that after the `Init` algorithm, we can imagine the state of the array to be essentially a random permutation. With probability about $1/256$, $S[02] = 00$ and $S[01] = X \neq 02$. In this case,

- The 1st step sends i to 01 , j to X , and swaps $S[01]$ and $S[X]$.
- The 2nd step sends i to 02 , leaves j as X , and swaps $S[02]$ and $S[X]$. Then $t := S[i] + S[j] = X$ and $y := S[t] = S[X] = 00$.

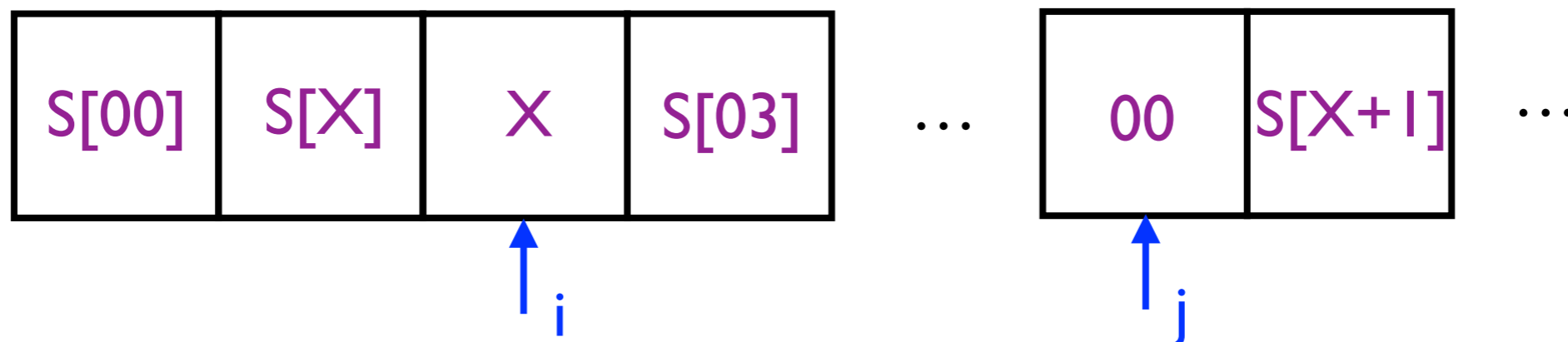
An Attack on RC4



Suppose that after the `Init` algorithm, we can imagine the state of the array to be essentially a random permutation. With probability about $1/256$, $S[02] = 00$ and $S[01] = X \neq 02$. In this case,

- The 1st step sends i to 01 , j to X , and swaps $S[01]$ and $S[X]$.
- The 2nd step sends i to 02 , leaves j as X , and swaps $S[02]$ and $S[X]$. Then $t := S[i] + S[j] = X$ and $y := S[t] = S[X] = 00$.

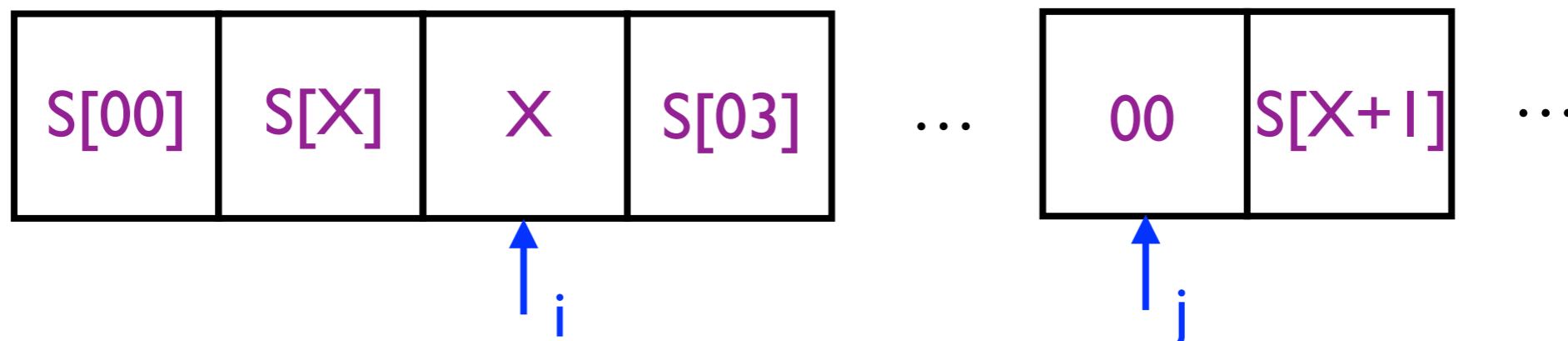
An Attack on RC4



Suppose that after the `Init` algorithm, we can imagine the state of the array to be essentially a random permutation. With probability about $1/256$, $S[02] = 00$ and $S[01] = X \neq 02$. In this case,

- The 1st step sends i to 01 , j to X , and swaps $S[01]$ and $S[X]$.
- The 2nd step sends i to 02 , leaves j as X , and swaps $S[02]$ and $S[X]$. Then $t := S[i] + S[j] = X$ and $y := S[t] = S[X] = 00$.

An Attack on RC4



Suppose that after the `Init` algorithm, we can imagine the state of the array to be essentially a random permutation. With probability about $1/256$, $S[02] = 00$ and $S[01] = X \neq 02$. In this case,

- The 1st step sends i to 01 , j to X , and swaps $S[01]$ and $S[X]$.
- The 2nd step sends i to 02 , leaves j as X , and swaps $S[02]$ and $S[X]$. Then $t := S[i] + S[j] = X$ and $y := S[t] = S[X] = 00$.

If $S[02] \neq 00$, t is essentially random, which leads to a $1/256$ chance that $y = 00$. The output 00 is twice as likely as random!

