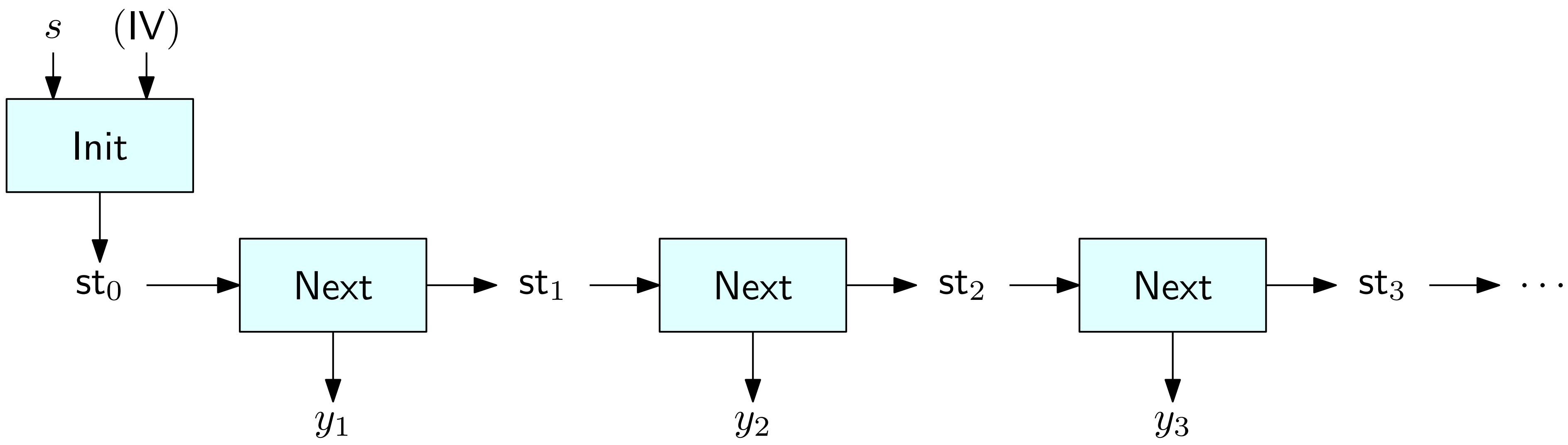
Stream ciphers (reminder)

A stream cipher is a pair of deterministic polynomial-time algorithms

- **Init:** takes a n-bit seed s, and possibly a n-bit *initialization vector* (IV), and outputs a *state* st
- **Next:** takes a state st and outputs a bit y and a new (updated) state st′

Idea: we can generate as many random bits as desired, by repeatedly calling Next



* In practice, **Next** can output multiple bits at once (e.g., a byte)

RC4

- Stands for Rivest Cipher 4
- Designed for performance in software

Ron Rivest (the R in RSA)

RC4

- Stands for Rivest Cipher 4
- Designed for performance in software
- Construction does **not** use (L)FSRs
- Very simple (fits one slide!)

Ron Rivest (the R in RSA)

RC4

- Stands for Rivest Cipher 4
- Designed for performance in software
- Construction does **not** use (L)FSRs
- Very simple (fits one slide!)
- **No longer considered secure (especially if misused)!**

  ... but still used in practice

WEP Encryption

Ron Rivest (the R in RSA)

RC4

- Stands for Rivest Cipher 4
- Designed for performance in software
- Construction does **not** use (L)FSRs
- Very simple (fits one slide!)
- **No longer considered secure (especially if misused)!**

  . . . but still used in practice

WEP Encryption

- We will see how to attack it

Ron Rivest (the R in RSA)

RC4

The state consists of:

- An array S of 256 bytes, which will always be a permutation of $\{0, \ldots, 255\}$
- A pair of integers $i, j \in \{0, \ldots, 255\}$

RC4

The state consists of:

- An array S of 256 bytes, which will always be a permutation of $\{0, \dots, 255\}$
- A pair of integers $i, j \in \{0, \dots, 255\}$

Init(k : array of 16 bytes):

- $S \leftarrow [0, 1, 2, \dots, 255]$
- $k \leftarrow \underbrace{k \,\|\, k \,\|\, \dots \,\|\, k}_{16 \text{ times}}$
- $j \leftarrow 0$
- For $i \leftarrow 0, 1, \dots, 255$:
  - $j \leftarrow j + S[i] + k[i] \pmod{256}$
  - Swap $S[i]$ and $S[j]$
- Return $\langle S, i = 0, j = 0 \rangle$

RC4

The state consists of:

- An array S of 256 bytes, which will always be a permutation of $\{0, \ldots, 255\}$
- A pair of integers $i, j \in \{0, \ldots, 255\}$

Init(k : array of 16 bytes):

- $S \leftarrow [0, 1, 2, \ldots, 255]$
- $k \leftarrow \underbrace{k \,\|\, k \,\|\, \ldots \,\|\, k}_{\text{16 times}}$
- $j \leftarrow 0$
- For $i \leftarrow 0, 1, \ldots, 255$:
  - $j \leftarrow j + S[i] + k[i] \pmod{256}$
  - Swap $S[i]$ and $S[j]$
- Return $\langle S, i = 0, j = 0 \rangle$

Next(st $= \langle S, i, j \rangle$): (returns a byte)

- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return the byte y and the new state st$' = \langle S, i, j \rangle$

[Demo]

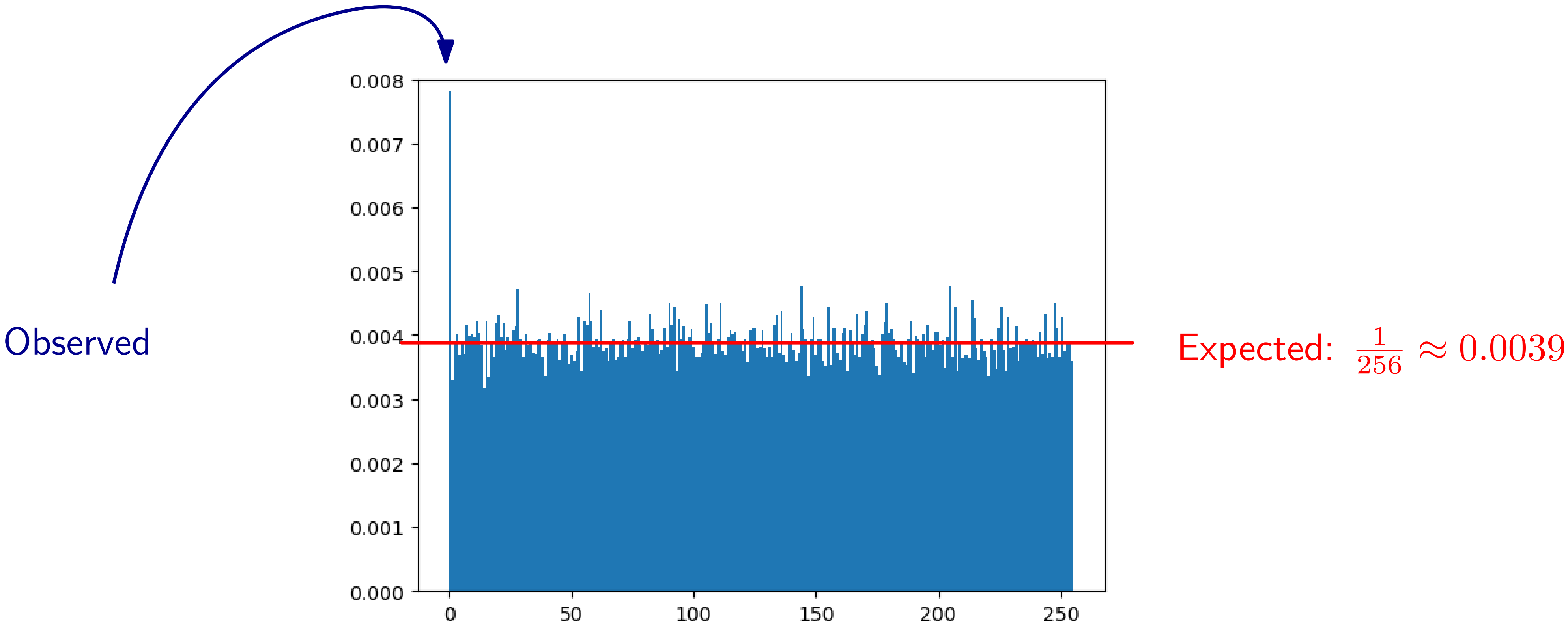
Test vectors

```
Key length: 128 bits.
key: 0x0102030405060708090a0b0c0d0e0f10

DEC    0 HEX    0:  9a c7 cc 9a  60 9d 1e f7   b2 93 28 99  cd e4 1b 97
DEC   16 HEX   10:  52 48 c4 95  90 14 12 6a   6e 8a 84 f1  1d 1a 9e 1c
DEC  240 HEX   f0:  06 59 02 e4  b6 20 f6 cc   36 c8 58 9f  66 43 2f 2b
DEC  256 HEX  100:  d3 9d 56 6b  c6 bc e3 01   07 68 15 15  49 f3 87 3f
DEC  496 HEX  1f0:  b6 d1 e6 c4  a5 e4 77 1c   ad 79 53 8d  f2 95 fb 11
DEC  512 HEX  200:  c6 8c 1d 5c  55 9a 97 41   23 df 1d bc  52 a4 3b 89
DEC  752 HEX  2f0:  c5 ec f8 8d  e8 97 fd 57   fe d3 01 70  1b 82 a2 59
DEC  768 HEX  300:  ec cb e1 3d  e1 fc c9 1c   11 a0 b2 6c  0b c8 fa 4d
DEC 1008 HEX  3f0:  e7 a7 25 74  f8 78 2a e2   6a ab cf 9e  bc d6 60 65
DEC 1024 HEX  400:  bd f0 32 4e  60 83 dc c6   d3 ce dd 3c  a8 c5 3c 16
DEC 1520 HEX  5f0:  b4 01 10 c4  19 0b 56 22   a9 61 16 b0  01 7e d2 97
DEC 1536 HEX  600:  ff a0 b5 14  64 7e c0 4f   63 06 b8 92  ae 66 11 81
DEC 2032 HEX  7f0:  d0 3d 1b c0  3c d3 3d 70   df f9 fa 5d  71 96 3e bd
DEC 2048 HEX  800:  8a 44 12 64  11 ea a7 8b   d5 1e 8d 87  a8 87 9b f5
DEC 3056 HEX  bf0:  fa be b7 60  28 ad e2 d0   e4 87 22 e4  6c 46 15 a3
DEC 3072 HEX  c00:  c0 5d 88 ab  d5 03 57 f9   35 a6 3c 59  ee 53 76 23
DEC 4080 HEX  ff0:  ff 38 26 5c  16 42 c1 ab   e8 d3 c2 fe  5e 57 2b f8
DEC 4096 HEX 1000:  a3 6a 4c 30  1a e8 ac 13   61 0c cb c1  22 56 ca cc
```

Output bias

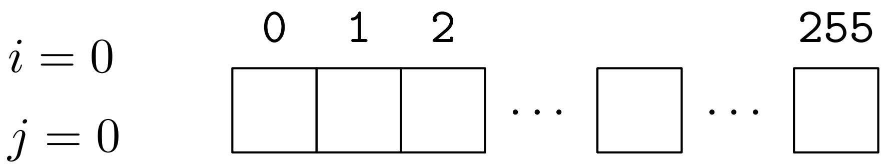
Empirical distribution of the value of the 2nd output byte over 50000 samples (with keys chosen u.a.r.)



There is a bias towards 0 in the second byte output by RC4 (about twice as likely to be 0)

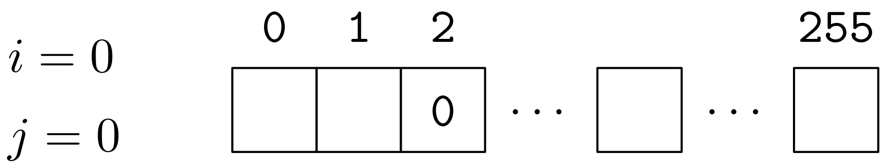
Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \ldots, 255\}$



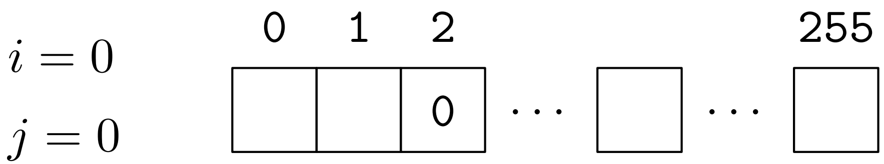
Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \ldots, 255\}$
- With probability $\approx \frac{1}{256}$ we have $S[2] = 0$



Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \ldots, 255\}$
- With probability $\approx \frac{1}{256}$ we have $S[2] = 0$

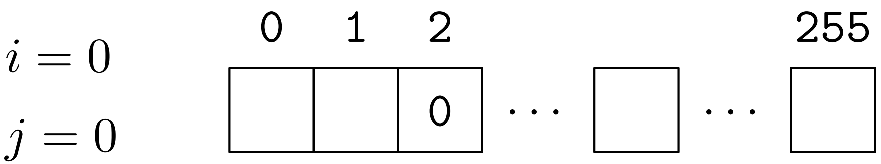


Next(st $= \langle S, i, j \rangle$**):** (returns a byte)

- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return the byte y and the new state st$' = \langle S, i, j \rangle$

Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \ldots, 255\}$
- With probability $\approx \frac{1}{256}$ we have $S[2] = 0$

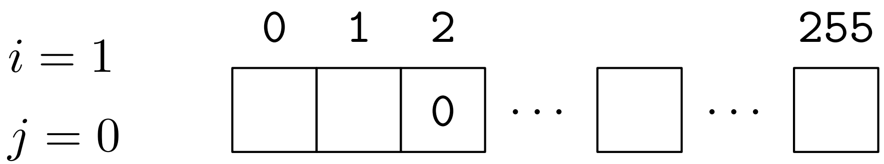


Next(st $= \langle S, i, j \rangle$**):** (returns a byte)

- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return the byte y and the new state st$' = \langle S, i, j \rangle$

Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \ldots, 255\}$
- With probability $\approx \frac{1}{256}$ we have $S[2] = 0$

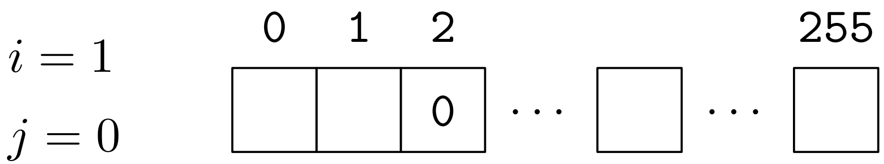


Next(st $= \langle S, i, j \rangle$)**:** (returns a byte)

- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return the byte y and the new state st$' = \langle S, i, j \rangle$

Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \ldots, 255\}$
- With probability $\approx \frac{1}{256}$ we have $S[2] = 0$

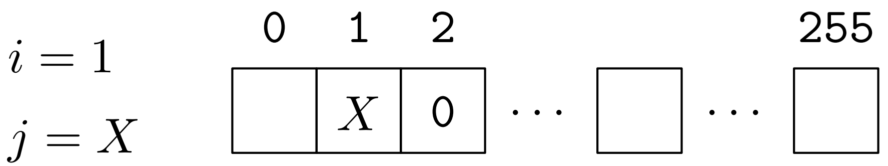


Next(st $= \langle S, i, j \rangle$**):** (returns a byte)

- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$ ←
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return the byte y and the new state st$' = \langle S, i, j \rangle$

Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \ldots, 255\}$
- With probability $\approx \frac{1}{256}$ we have $S[2] = 0$



Next(st $= \langle S, i, j \rangle$**):** (returns a byte)

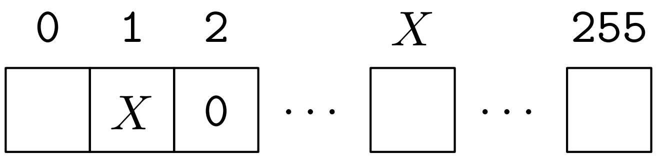
- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return the byte y and the new state st$' = \langle S, i, j \rangle$

Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \ldots, 255\}$
- With probability $\approx \frac{1}{256}$ we have $S[2] = 0$

$i = 1$

$j = X$

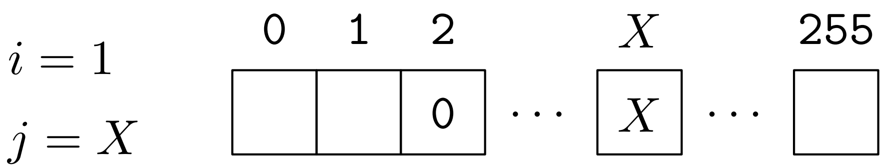


Next(st $= \langle S, i, j \rangle$**):** (returns a byte)

- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$ ←
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return the byte y and the new state st$' = \langle S, i, j \rangle$

Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \ldots, 255\}$
- With probability $\approx \frac{1}{256}$ we have $S[2] = 0$



Next(st $= \langle S, i, j \rangle$**):** (returns a byte)

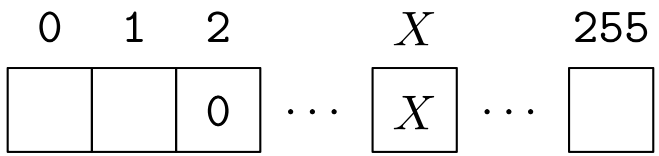
- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return the byte y and the new state st$' = \langle S, i, j \rangle$

Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \ldots, 255\}$
- With probability $\approx \frac{1}{256}$ we have $S[2] = 0$

$i = 1$

$j = X$



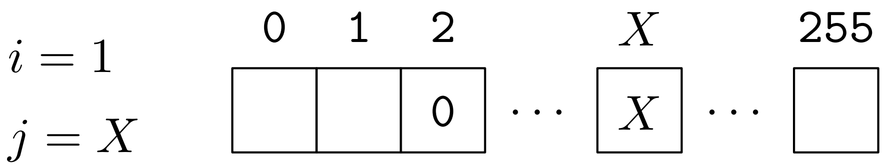
Next(st $= \langle S, i, j \rangle$)**:** (returns a byte)

- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return the byte y and the new state st$' = \langle S, i, j \rangle$

The rest of the code does not modify the state

Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \ldots, 255\}$
- With probability $\approx \frac{1}{256}$ we have $S[2] = 0$



Next(st $= \langle S, i, j \rangle$)**:** 2nd call (returns a byte)

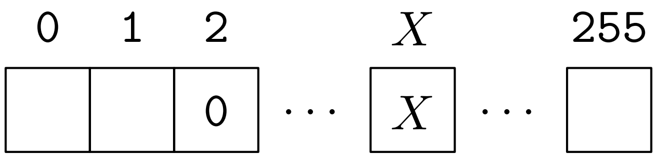
- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return the byte y and the new state st$' = \langle S, i, j \rangle$

Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \ldots, 255\}$
- With probability $\approx \frac{1}{256}$ we have $S[2] = 0$

$i = 2$

$j = X$

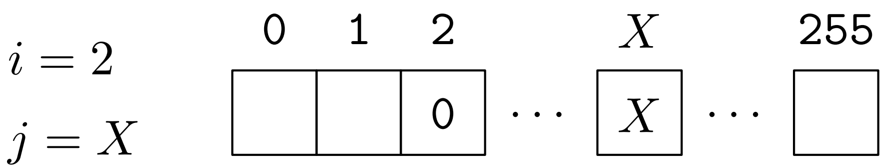


Next(st $= \langle S, i, j \rangle$)**:** 2nd call (returns a byte)

- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return the byte y and the new state st$' = \langle S, i, j \rangle$

Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \ldots, 255\}$
- With probability $\approx \frac{1}{256}$ we have $S[2] = 0$



Next(st $= \langle S, i, j \rangle$**):** 2nd call (returns a byte)

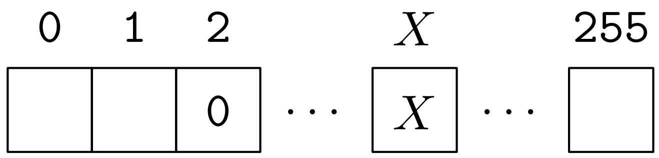
- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return the byte y and the new state st$' = \langle S, i, j \rangle$

Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \dots, 255\}$
- With probability $\approx \frac{1}{256}$ we have $S[2] = 0$

$i = 2$

$j = X$



Next(st $= \langle S, i, j \rangle$)**:** 2nd call (returns a byte)

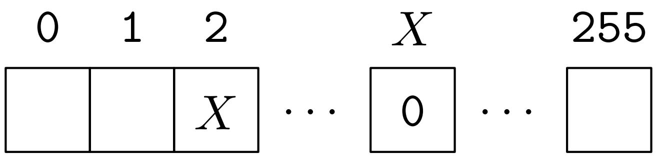
- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return the byte y and the new state st$' = \langle S, i, j \rangle$

Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \ldots, 255\}$
- With probability $\approx \frac{1}{256}$ we have $S[2] = 0$

$i = 2$

$j = X$



Next(st $= \langle S, i, j \rangle$)**:** 2nd call (returns a byte)

- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return the byte y and the new state st$' = \langle S, i, j \rangle$

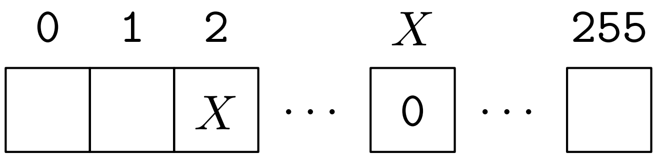
Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \dots, 255\}$
- With probability $\approx \frac{1}{256}$ we have $S[2] = 0$

$i = 2$

$j = X$

$t = X$



Next(st $= \langle S, i, j \rangle$)**:** 2nd call (returns a byte)

- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return the byte y and the new state st$' = \langle S, i, j \rangle$

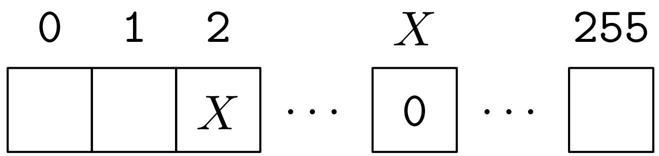
Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \ldots, 255\}$
- With probability $\approx \frac{1}{256}$ we have $S[2] = 0$

$i = 2$

$j = X$

$t = X$



Next(st $= \langle S, i, j \rangle$)**:** 2nd call (returns a byte)

- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return the byte y and the new state st$' = \langle S, i, j \rangle$

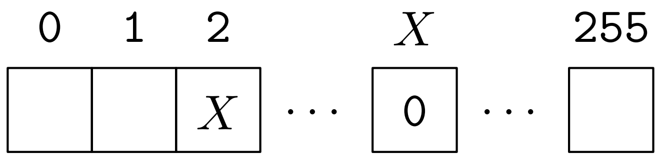
Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \ldots, 255\}$
- With probability $\approx \frac{1}{256}$ we have $S[2] = 0$

$i = 2$

$j = X$

$t = X$



Output byte $y = 0$

Next(st $= \langle S, i, j \rangle$**):** 2nd call (returns a byte)

- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return the byte y and the new state st$' = \langle S, i, j \rangle$

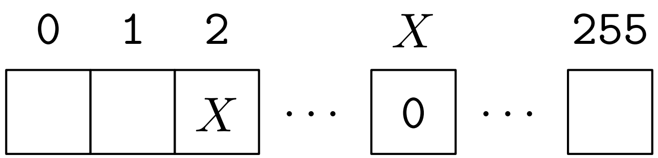
Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \ldots, 255\}$
- With probability $\approx \frac{1}{256}$ we have $S[2] = 0$

$i = 2$

$j = X$

$t = X$



Output byte $y = 0$

- With probability $\approx \frac{255}{256} \approx 1$ we have that $S[2]$ is distributed "uniformly at random" after 2 iterations

Next(st $= \langle S, i, j \rangle$**):** (returns a byte)

- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return the byte y and the new state st$' = \langle S, i, j \rangle$

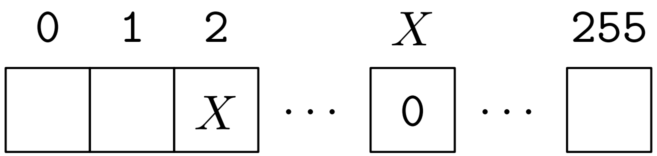
Output bias: analysis

- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \ldots, 255\}$
- With probability $\approx \frac{1}{256}$ we have $S[2] = 0$

$i = 2$

$j = X$

$t = X$



Output byte $y = 0$

- With probability $\approx \frac{255}{256} \approx 1$ we have that $S[2]$ is distributed "uniformly at random" after 2 iterations

Probability that the 2nd output byte is 0:

$\approx \frac{1}{256} + 1 \cdot \frac{1}{256} = \frac{2}{256}$

Next(st $= \langle S, i, j \rangle$**):** (returns a byte)

- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return the byte y and the new state st$' = \langle S, i, j \rangle$

Output bias

- The output bias is indicative of structural problems with RC4
- Other biases have been found in other bytes of the RC4 state
- Severe enough to allow recovery of plaintext from ciphertext when RC4 is used for encryption!

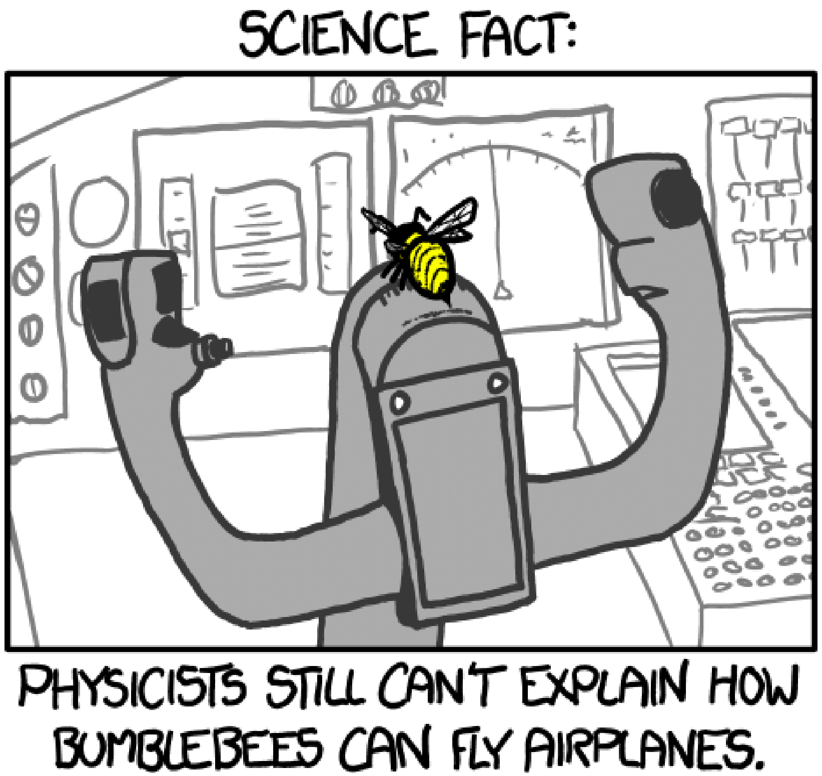
Output bias

- The output bias is indicative of structural problems with RC4
- Other biases have been found in other bytes of the RC4 state
- Severe enough to allow recovery of plaintext from ciphertext when RC4 is used for encryption!

In summary: Do not use RC4!

RC4 and IVs

RC4 is **not** designed to take an IV . . . but programmers don’t know it and use an IV anyway



xkcd.com

RC4 and IVs

RC4 is **not** designed to take an IV

In practice an IV of some length ℓ (in bytes) is often used, together with a key k' of $16 - \ell$ bytes

$$k = \mathsf{IV} \,\|\, k'$$

RC4 and IVs

RC4 is **not** designed to take an IV

In practice an IV of some length ℓ (in bytes) is often used, together with a key k' of $16 - \ell$ bytes

$$k = \text{IV} \,\|\, k'$$

In WEP:

- 3-byte IV, 13 bytes key

RC4 and IVs

RC4 is **not** designed to take an IV

In practice an IV of some length ℓ (in bytes) is often used, together with a key k' of $16 - \ell$ bytes

$$k = \mathsf{IV} \,\|\, k'$$

In WEP:

- 3-byte IV, 13 bytes key
- Key recovery attack!

RC4 and IVs

RC4 is **not** designed to take an IV

In practice an IV of some length ℓ (in bytes) is often used, together with a key k' of $16 - \ell$ bytes

$$k = \mathsf{IV} \,\|\, k'$$

In WEP:

- 3-byte IV, 13 bytes key
- Key recovery attack!
- We show a simplified attack that recovers the first byte of the key (i.e., $k[3]$)

Key recovery attack

- Recall that **IV**s are not kept secret!

Key recovery attack

- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)

this is just one possibility
(attacks for other combinations are also known)

Key recovery attack

- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility
(attacks for other combinations are also known)

	0	1	2	3	
k	3	255	X	Ψ	$\cdots$

Key recovery attack

- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility
(attacks for other combinations are also known)

> **Init(k : array of 16 bytes):**
>
> - $S \leftarrow [0, 1, 2, \ldots, 255]$
> - $k \leftarrow \underbrace{k \,\|\, k \,\|\, \ldots \,\|\, k}_{\text{16 times}}$
> - $j \leftarrow 0$
> - For $i \leftarrow 0, 1, \ldots, 255$:
>   - $j \leftarrow j + S[i] + k[i] \pmod{256}$
>   - Swap $S[i]$ and $S[j]$
> - Return $\langle S, i = 0, j = 0 \rangle$

	0	1	2	3	
k	3	255	X	Ψ	$\cdots$

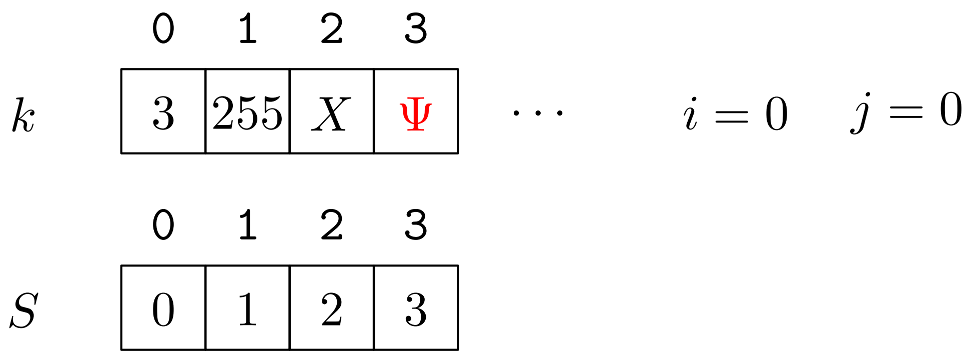
Key recovery attack

- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility
(attacks for other combinations are also known)

Init(k : array of 16 bytes):

- $S \leftarrow [0, 1, 2, \ldots, 255]$
- $k \leftarrow \underbrace{k \,\|\, k \,\|\, \ldots \,\|\, k}_{\text{16 times}}$
- $j \leftarrow 0$
- For $i \leftarrow 0, 1, \ldots, 255$:
    - $j \leftarrow j + S[i] + k[i] \pmod{256}$
    - Swap $S[i]$ and $S[j]$
- Return $\langle S, i = 0, j = 0 \rangle$



Key recovery attack

- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility
(attacks for other combinations are also known)

Init(k : array of 16 bytes):

- $S \leftarrow [0, 1, 2, \ldots, 255]$
- $k \leftarrow \underbrace{k \,\|\, k \,\|\, \ldots \,\|\, k}_{\text{16 times}}$
- $j \leftarrow 0$
- For $i \leftarrow 0, 1, \ldots, 255$:
  - $j \leftarrow j + S[i] + k[i] \pmod{256}$
  - Swap $S[i]$ and $S[j]$ ←
- Return $\langle S, i = 0, j = 0 \rangle$

	0	1	2	3	
k	3	255	X	Ψ	$\cdots$

$i = 0 \quad j = 3$

	0	1	2	3
S	0	1	2	3

Key recovery attack

- Recall that **IVs** are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility
(attacks for other combinations are also known)

Init(k : array of 16 bytes):

- $S \leftarrow [0, 1, 2, \ldots, 255]$
- $k \leftarrow \underbrace{k \,\|\, k \,\|\, \ldots \,\|\, k}_{\text{16 times}}$
- $j \leftarrow 0$
- For $i \leftarrow 0, 1, \ldots, 255$:
  - $j \leftarrow j + S[i] + k[i] \pmod{256}$
  - Swap $S[i]$ and $S[j]$
- Return $\langle S, i = 0, j = 0 \rangle$

	0	1	2	3	
k	3	255	X	Ψ	$\cdots$

$i = 1 \quad j = 3$

	0	1	2	3
S	3	1	2	0

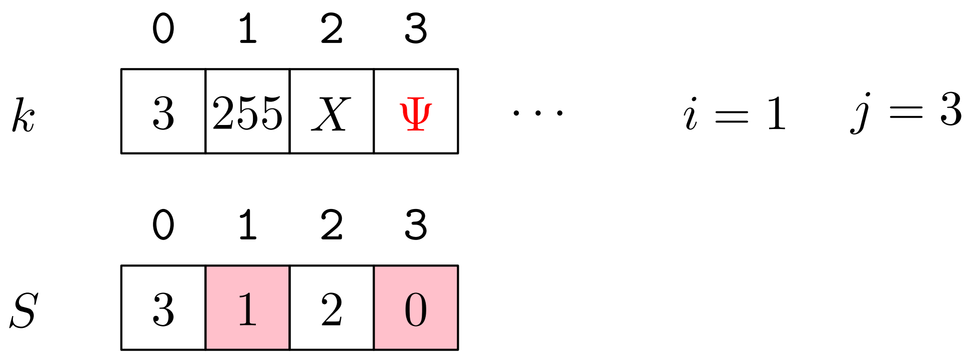
Key recovery attack

- Recall that **IVs** are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility
(attacks for other combinations are also known)

Init(k : array of 16 bytes):

- $S \leftarrow [0, 1, 2, \ldots, 255]$
- $k \leftarrow \underbrace{k \,\|\, k \,\|\, \ldots \,\|\, k}_{\text{16 times}}$
- $j \leftarrow 0$
- For $i \leftarrow 0, 1, \ldots, 255$:
  - $j \leftarrow j + S[i] + k[i] \pmod{256}$
  - Swap $S[i]$ and $S[j]$
- Return $\langle S, i = 0, j = 0 \rangle$



Key recovery attack

- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility
(attacks for other combinations are also known)

Init(k : array of 16 bytes):

- $S \leftarrow [0, 1, 2, \ldots, 255]$
- $k \leftarrow \underbrace{k \,\|\, k \,\|\, \ldots \,\|\, k}_{\text{16 times}}$
- $j \leftarrow 0$
- For $i \leftarrow 0, 1, \ldots, 255$:
  - $j \leftarrow j + S[i] + k[i] \pmod{256}$
  - Swap $S[i]$ and $S[j]$
- Return $\langle S, i = 0, j = 0 \rangle$

	0	1	2	3	
k	3	255	X	Ψ	$\cdots$

$i = 2 \quad j = 3$

	0	1	2	3
S	3	0	2	1

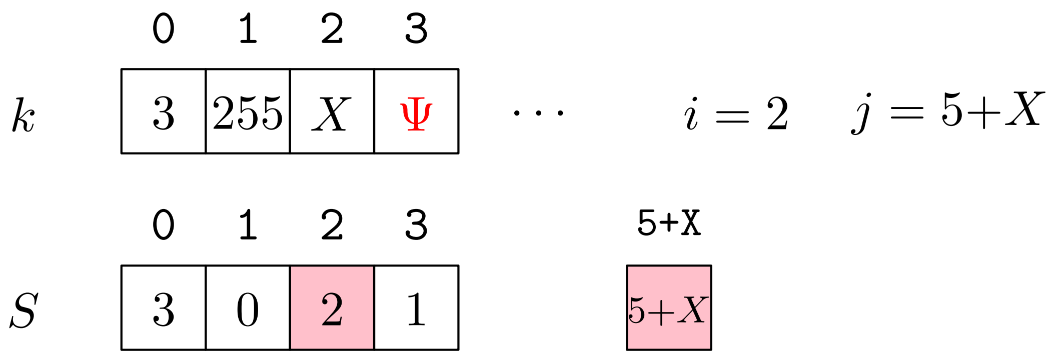
Key recovery attack

- Recall that **IVs** are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility
(attacks for other combinations are also known)

Init(k : array of 16 bytes):

- $S \leftarrow [0, 1, 2, \ldots, 255]$
- $k \leftarrow \underbrace{k \,\|\, k \,\|\, \ldots \,\|\, k}_{\text{16 times}}$
- $j \leftarrow 0$
- For $i \leftarrow 0, 1, \ldots, 255$:
  - $j \leftarrow j + S[i] + k[i] \pmod{256}$
  - Swap $S[i]$ and $S[j]$ ←
- Return $\langle S, i = 0, j = 0 \rangle$



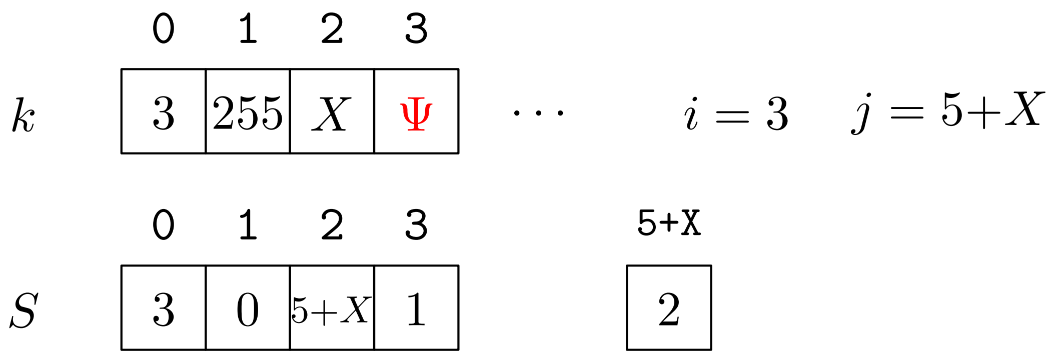
Key recovery attack

- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X\rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility
(attacks for other combinations are also known)

Init(k : array of 16 bytes):

- $S \leftarrow [0, 1, 2, \ldots, 255]$
- $k \leftarrow \underbrace{k \,\|\, k \,\|\, \ldots \,\|\, k}_{\text{16 times}}$
- $j \leftarrow 0$
- For $i \leftarrow 0, 1, \ldots, 255$:
  - $j \leftarrow j + S[i] + k[i] \pmod{256}$
  - Swap $S[i]$ and $S[j]$
- Return $\langle S, i = 0, j = 0\rangle$



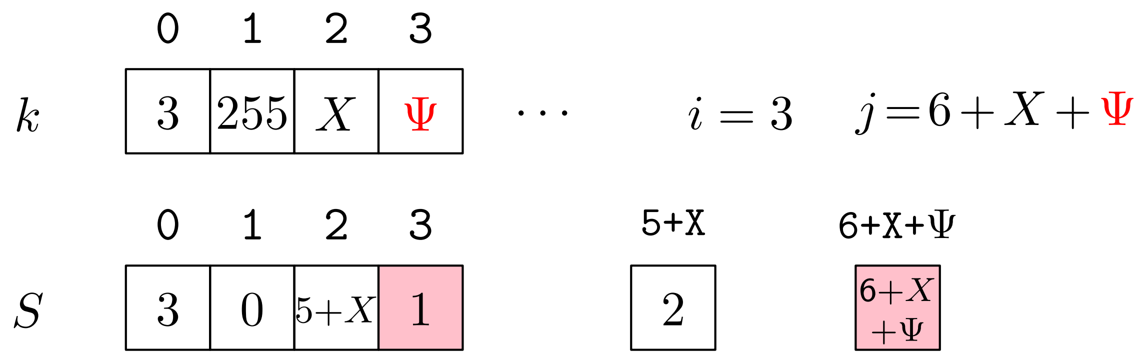
Key recovery attack

- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility
(attacks for other combinations are also known)

Init(k : array of 16 bytes):

- $S \leftarrow [0, 1, 2, \ldots, 255]$
- $k \leftarrow \underbrace{k \,\|\, k \,\|\, \ldots \,\|\, k}_{\text{16 times}}$
- $j \leftarrow 0$
- For $i \leftarrow 0, 1, \ldots, 255$:
  - $j \leftarrow j + S[i] + k[i] \pmod{256}$
  - Swap $S[i]$ and $S[j]$
- Return $\langle S, i = 0, j = 0 \rangle$



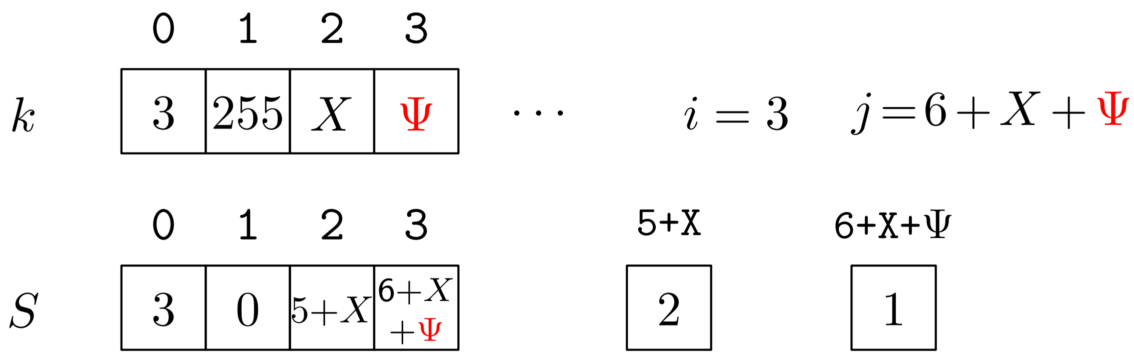
Key recovery attack

- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility
(attacks for other combinations are also known)

Init(k : array of 16 bytes):

- $S \leftarrow [0, 1, 2, \ldots, 255]$
- $k \leftarrow \underbrace{k \,\|\, k \,\|\, \ldots \,\|\, k}_{\text{16 times}}$
- $j \leftarrow 0$
- For $i \leftarrow 0, 1, \ldots, 255$:
  - $j \leftarrow j + S[i] + k[i] \pmod{256}$
  - Swap $S[i]$ and $S[j]$
- Return $\langle S, i = 0, j = 0 \rangle$



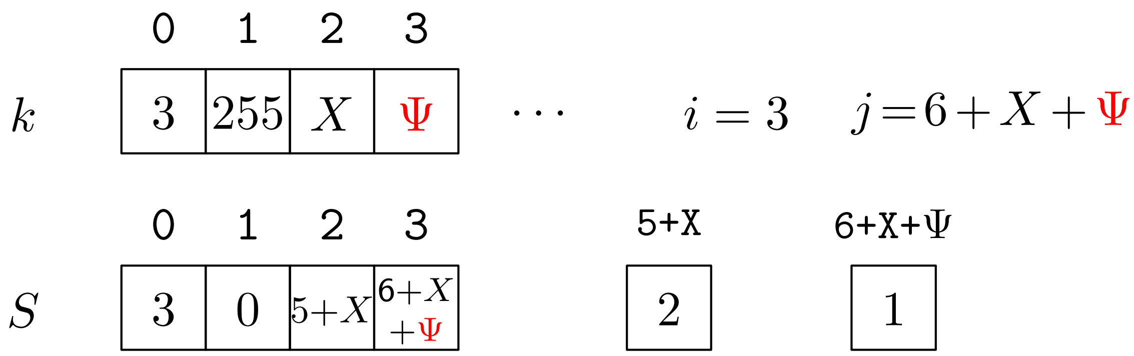
Key recovery attack

- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility
(attacks for other combinations are also known)

Init(k : array of 16 bytes):

- $S \leftarrow [0, 1, 2, \ldots, 255]$
- $k \leftarrow \underbrace{k \,\|\, k \,\|\, \ldots \,\|\, k}_{\text{16 times}}$
- $j \leftarrow 0$
- For $i \leftarrow 0, 1, \ldots, 255$:
  - $j \leftarrow j + S[i] + k[i] \pmod{256}$
  - Swap $S[i]$ and $S[j]$
- Return $\langle S, i = 0, j = 0 \rangle$



With probability $\approx 5\%$, $S[3]$ is not modified in the remaining iterations of Init

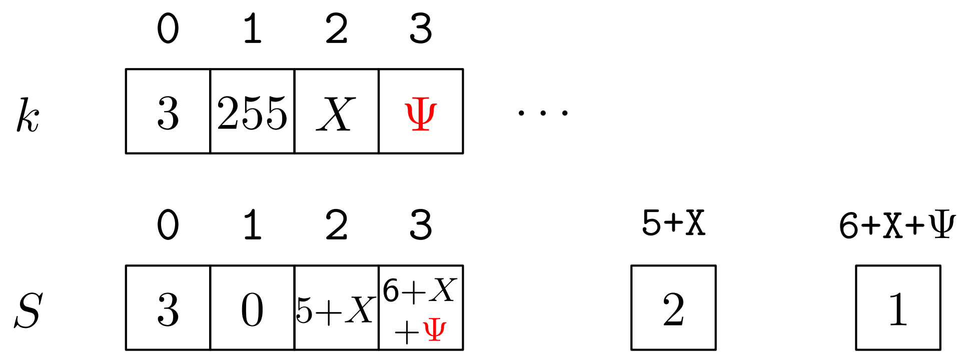
Key recovery attack

- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility
(attacks for other combinations are also known)

Next(st $= \langle S, i, j \rangle$**):**

- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return y and st$' = \langle S, i, j \rangle$



With probability $\approx 5\%$, $S[3]$ is not modified in the remaining iterations of Init

What's the first byte output by Next (when $i = j = 0$)?

Key recovery attack

- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility
(attacks for other combinations are also known)

Next(st $= \langle S, i, j \rangle$**):**

- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return y and st$' = \langle S, i, j \rangle$

$i = 1$

	0	1	2	3	
k	3	255	X	Ψ	$\cdots$

	0	1	2	3	5+X	6+X+Ψ
S	3	0	$5+X$	$6+X+\Psi$	2	1

With probability $\approx 5\%$, $S[3]$ is not modified in the remaining iterations of Init

What's the first byte output by Next (when $i = j = 0$)?

Key recovery attack

- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X\rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

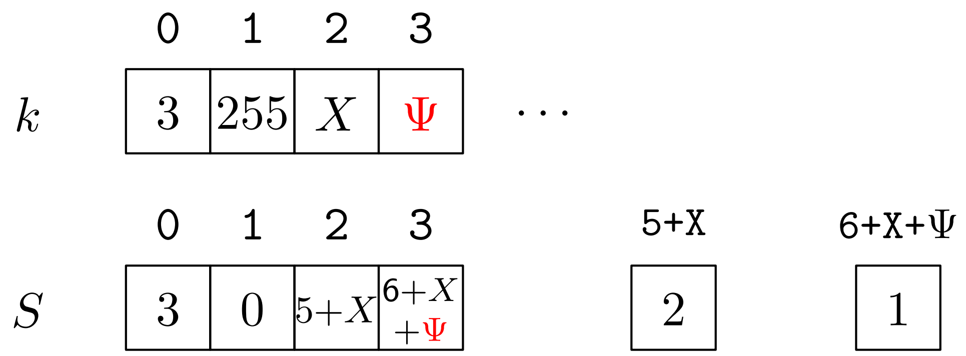
this is just one possibility
(attacks for other combinations are also known)

Next(st $= \langle S, i, j\rangle$**):**

- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return y and st$' = \langle S, i, j\rangle$

$i = 1$

$j = 0$



With probability $\approx 5\%$, $S[3]$ is not modified in the remaining iterations of Init

What's the first byte output by Next (when $i = j = 0$)?

Key recovery attack

- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility
(attacks for other combinations are also known)

Next(st $= \langle S, i, j \rangle$**):**

- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return y and st$' = \langle S, i, j \rangle$

$i = 1$

$j = 0$

$t = 3$

	0	1	2	3	
k	3	255	X	Ψ	$\cdots$

	0	1	2	3	5+X	6+X+Ψ
S	3	0	$5+X$	$6+X+\Psi$	2	1

With probability $\approx 5\%$, $S[3]$ is not modified in the remaining iterations of Init

What's the first byte output by Next (when $i = j = 0$)?

Key recovery attack

- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility
(attacks for other combinations are also known)

Next(st $= \langle S, i, j \rangle$)**:**

- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return y and st$' = \langle S, i, j \rangle$

$i = 1$

$j = 0$

$t = 3$

$y = S[3]$

	0	1	2	3	
k	3	255	X	Ψ	$\cdots$

	0	1	2	3	5+X	6+X+Ψ
S	3	0	$5{+}X$	$6{+}X{+}\Psi$	2	1

With probability $\approx 5\%$, $S[3]$ is not modified in the remaining iterations of Init

What's the first byte output by Next (when $i = j = 0$)?

Key recovery attack

- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility
(attacks for other combinations are also known)

Next(st $= \langle S, i, j \rangle$**):**

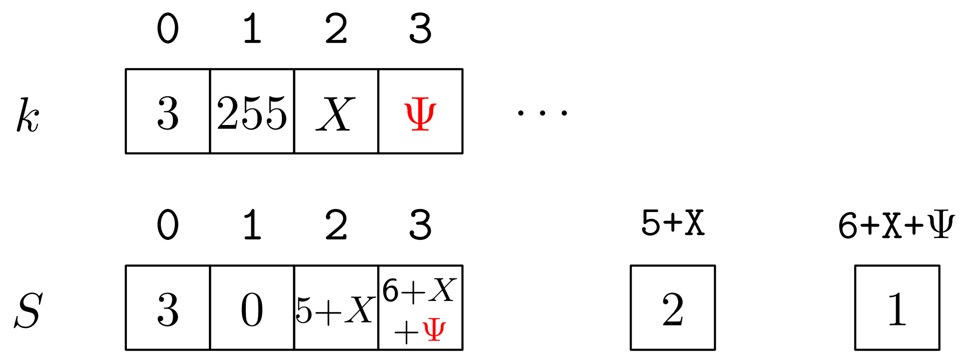
- $i \leftarrow i + 1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap $S[i]$ and $S[j]$
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$
- Return y and st$' = \langle S, i, j \rangle$

$i = 1$

$j = 0$

$t = 3$

$y = S[3]$



With probability $\approx 5\%$, $S[3]$ is not modified in the remaining iterations of Init

What's the first byte output by Next (when $i = j = 0$)?

$$6 + X + \Psi$$

Key recovery attack

- 5% of the time the adversary sees $6 + X + \Psi$
- Since X is known (it is part of the IV), the adversary can recover Ψ

Key recovery attack

- 5% of the time the adversary sees $6 + X + \Psi$
- Since X is known (it is part of the IV), the adversary can recover Ψ
- Quite far from uniform: $\frac{1}{256} \approx 0.4\%$

Key recovery attack

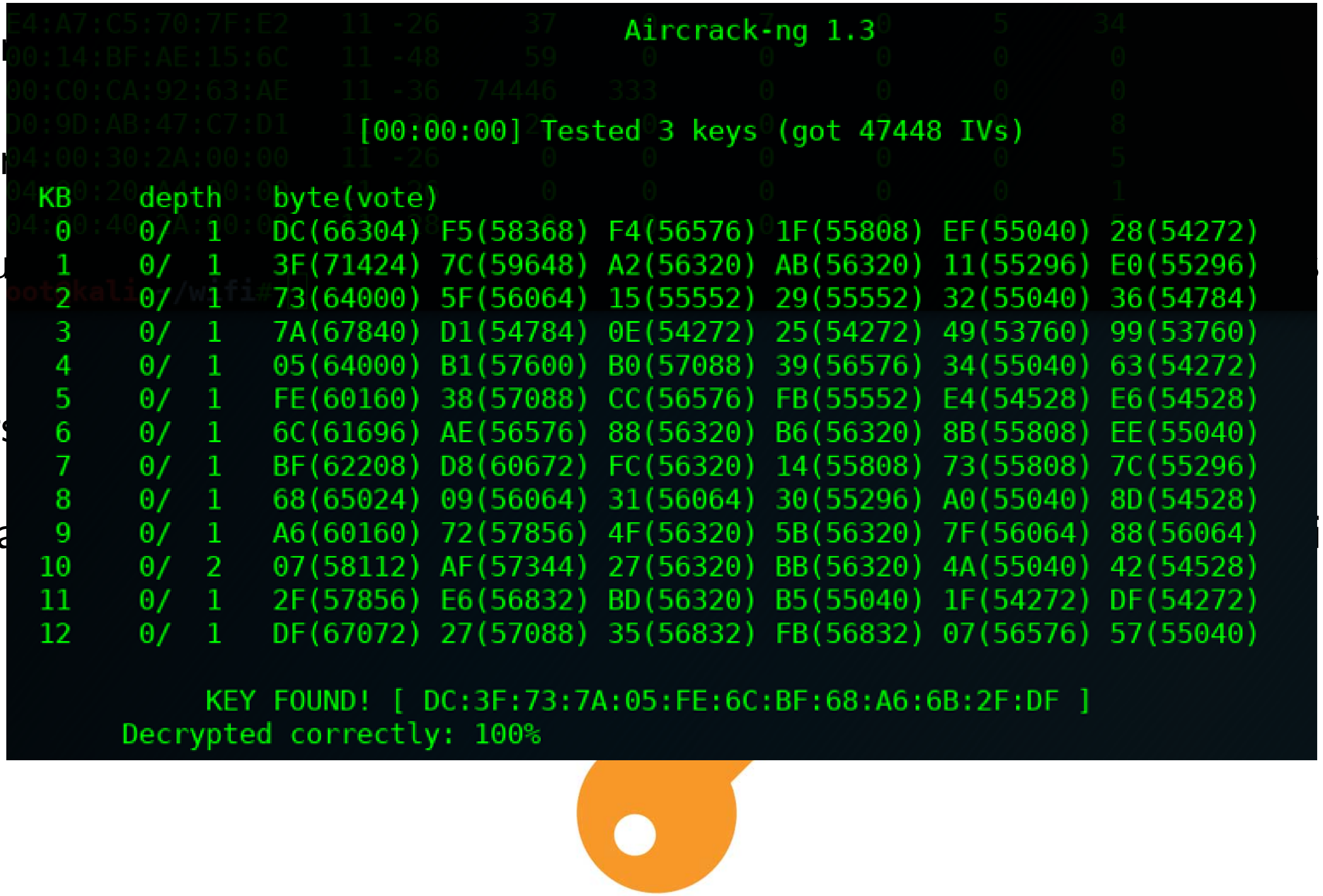
- 5% of the time the adversary sees $6 + X + \Psi$
- Since X is known (it is part of the IV), the adversary can recover Ψ
- Quite far from uniform: $\frac{1}{256} \approx 0.4\%$
- Wait for a sufficiently large number of IVs for which the first byte of the key is leaked (with some probability)
- Guess the first byte of the key (with high confidence)

Key recovery attack

- 5% of the time the adversary sees $6 + X + \Psi$
- Since X is known (it is part of the IV), the adversary can recover Ψ
- Quite far from uniform: $\frac{1}{256} \approx 0.4\%$
- Wait for a sufficiently large number of IVs for which the first byte of the key is leaked (with some probability)
- Guess the first byte of the key (with high confidence)
- Repeat similar attacks to extract the next byte of the key, until the whole key is reconstructed

Key recovery attack

- 5% of the time the adversary sees $6 + X + \color{red}{\Psi}$
- Since X is k
- Quite far fro
- Wait for a su … leaked (with some probability)
- Guess the fir
- Repeat simila … is reconstructed



ChaCha20

Introduced in 2008. Secure replacement for RC4

Takes a 256-bit key k and a 64-bit IV

Daniel J. Bernstein

ChaCha20

Introduced in 2008. Secure replacement for RC4

Takes a 256-bit key k and a 64-bit IV

Relies on addition, rotations, and XOR of 32-bit words
(all of which typically require just one assembly instruction)

Daniel J. Bernstein

ChaCha20

Introduced in 2008. Secure replacement for RC4

Daniel J. Bernstein

Takes a 256-bit key k and a 64-bit IV

Relies on addition, rotations, and XOR of 32-bit words
(all of which typically require just one assembly instruction)

The core of ChaCha20 is a fixed permutation $P : \{0,1\}^{512} \to \{0,1\}^{512}$ on 512-bit strings

The permutation P is used to construct a keyed function with a 256-bit key, 128-bit inputs and 512-bit outputs

$$F_k(x) = P(\text{constant} \,\|\, k \,\|\, x) \boxplus (\text{constant} \,\|\, k \,\|\, x)$$

ChaCha20

Introduced in 2008. Secure replacement for RC4

Daniel J. Bernstein

Takes a 256-bit key k and a 64-bit IV

Relies on addition, rotations, and XOR of 32-bit words
(all of which typically require just one assembly instruction)

The core of ChaCha20 is a fixed permutation $P : \{0,1\}^{512} \to \{0,1\}^{512}$ on 512-bit strings

The permutation P is used to construct a keyed function with a 256-bit key, 128-bit inputs and 512-bit outputs

$$F_k(x) = P(\text{constant} \,\|\, k \,\|\, x) \boxplus (\text{constant} \,\|\, k \,\|\, x)$$

$\boxplus$ denotes word-wise modular addition (of 32-bit words)

ChaCha20

Introduced in 2008. Secure replacement for RC4

Daniel J. Bernstein

Takes a 256-bit key k and a 64-bit IV

Relies on addition, rotations, and XOR of 32-bit words
(all of which typically require just one assembly instruction)

The core of ChaCha20 is a fixed permutation $P : \{0,1\}^{512} \to \{0,1\}^{512}$ on 512-bit strings

The permutation P is used to construct a keyed function with a 256-bit key, 128-bit inputs and 512-bit outputs

$$F_k(x) = P(\text{constant} \,\|\, k \,\|\, x) \boxplus (\text{constant} \,\|\, k \,\|\, x)$$

$\boxplus$ denotes word-wise modular addition (of 32-bit words)

Output stream:

$$F_k(\text{IV} \,\|\, \langle 0 \rangle), F_k(\text{IV} \,\|\, \langle 1 \rangle), F_k(\text{IV} \,\|\, \langle 2 \rangle), \ldots$$

$\langle i \rangle$ = binary encoding of i with 64 bits

ChaCha20

Introduced in 2008. Secure replacement for RC4

Daniel J. Bernstein

Takes a 256-bit key k and a 64-bit IV

Relies on addition, rotations, and XOR of 32-bit words
(all of which typically require just one assembly instruction)

The core of ChaCha20 is a fixed permutation $P : \{0,1\}^{512} \to \{0,1\}^{512}$ on 512-bit strings

The permutation P is used to construct a keyed function with a 256-bit key, 128-bit inputs and 512-bit outputs

$$F_k(x) = P(\text{constant} \,\|\, k \,\|\, x) \boxplus (\text{constant} \,\|\, k \,\|\, x)$$

$\boxplus$ denotes word-wise modular addition (of 32-bit words)

Output stream:

$$F_k(\text{IV} \,\|\, \langle 0 \rangle), F_k(\text{IV} \,\|\, \langle 1 \rangle), F_k(\text{IV} \,\|\, \langle 2 \rangle), \ldots$$

$\langle i \rangle$ = binary encoding of i with 64 bits

Not patented. Several public domain implementations available

Block Ciphers

A block cipher is. . .

Block Ciphers

A block cipher is. . . just another name for a (possibly strong) pseudorandom permutation

$$F : \{0,1\}^{\ell_{key}(n)} \times \{0,1\}^{\ell_{in}(n)} \rightarrow \{0,1\}^{\ell_{out}(n)}$$

Block Ciphers

A block cipher is. . . just another name for a (possibly strong) pseudorandom permutation

$$F : \{0,1\}^{\ell_{key}(n)} \times \{0,1\}^{\ell_{in}(n)} \rightarrow \{0,1\}^{\ell_{out}(n)}$$

You can think of block ciphers as *practical constructions* of (candidate) pseudorandom permutations

Block Ciphers

A block cipher is. . . just another name for a (possibly strong) pseudorandom permutation

$$F : \{0,1\}^{\ell_{key}(n)} \times \{0,1\}^{\ell_{in}(n)} \rightarrow \{0,1\}^{\ell_{out}(n)}$$

You can think of block ciphers as *practical constructions* of (candidate) pseudorandom permutations

Block ciphers typically only support a specific set of key/block lengths

We consider $\ell_{key}(n) = n$ and $\ell_{in}(n) = \ell_{out}(n) = n$

n is called the **block length** of F

Block Ciphers

A block cipher is. . . just another name for a (possibly strong) pseudorandom permutation

$$F : \{0,1\}^{\ell_{key}(n)} \times \{0,1\}^{\ell_{in}(n)} \to \{0,1\}^{\ell_{out}(n)}$$

You can think of block ciphers as *practical constructions* of (candidate) pseudorandom permutations

Block ciphers typically only support a specific set of key/block lengths

We consider $\ell_{key}(n) = n$ and $\ell_{in}(n) = \ell_{out}(n) = n$

n is called the **block length** of F

We assume for simplicity that the message m to be encrypted can be split into blocks $m_1, m_2, m_3, \ldots$ of lengths exactly n

$m =$

m_1	m_2	m_3	$\cdots$			

Block Ciphers

A block cipher is... just another name for a (possibly strong) pseudorandom permutation

$$F : \{0,1\}^{\ell_{key}(n)} \times \{0,1\}^{\ell_{in}(n)} \to \{0,1\}^{\ell_{out}(n)}$$

You can think of block ciphers as *practical constructions* of (candidate) pseudorandom permutations

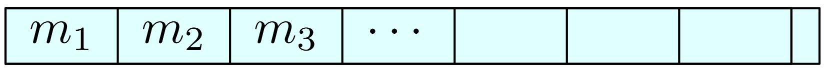
Block ciphers typically only support a specific set of key/block lengths

We consider $\ell_{key}(n) = n$ and $\ell_{in}(n) = \ell_{out}(n) = n$

n is called the **block length** of F

We assume for simplicity that the message m to be encrypted can be split into blocks $m_1, m_2, m_3, \ldots$ of lengths exactly n

$m =$



What if the length of m is not a multiple of n?

Block Ciphers

A block cipher is. . . just another name for a (possibly strong) pseudorandom permutation

$$F : \{0,1\}^{\ell_{key}(n)} \times \{0,1\}^{\ell_{in}(n)} \to \{0,1\}^{\ell_{out}(n)}$$

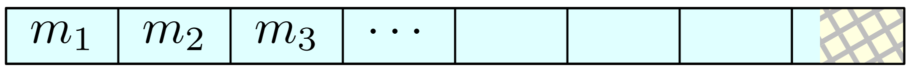
You can think of block ciphers as *practical constructions* of (candidate) pseudorandom permutations

Block ciphers typically only support a specific set of key/block lengths

We consider $\ell_{key}(n) = n$ and $\ell_{in}(n) = \ell_{out}(n) = n$

n is called the **block length** of F

We assume for simplicity that the message m to be encrypted can be split into blocks $m_1, m_2, m_3, \ldots$ of lengths exactly n

$m =$ 

What if the length of m is not a multiple of n? Padding (with care)

Block Ciphers

Recall that we can always build a stream cipher from a block cipher

For example:

Init(s, IV):

- Output $(s, \text{IV}, 0)$

Next(st):

- Unpack the state in $(s, \text{IV}, \langle i \rangle)$
- Output the n bits $F_s(\text{IV} \parallel \langle i \rangle)$ and the new state $(s, \text{IV}, \langle i+1 \rangle)$

Block Ciphers

Recall that we can always build a stream cipher from a block cipher

For example:

$3n/4$ bits

Init(s, IV):

- Output $(s, \mathsf{IV}, 0)$

Next(st):

- Unpack the state in $(s, \mathsf{IV}, \langle i \rangle)$
- Output the n bits $F_s(\mathsf{IV} \,\|\, \langle i \rangle)$ and the new state $(s, \mathsf{IV}, \langle i+1 \rangle)$

Block Ciphers

Recall that we can always build a stream cipher from a block cipher

For example:

$3n/4$ bits

Init(s, IV):

- Output $(s, \text{IV}, 0)$

Next(st):

- Unpack the state in $(s, \text{IV}, \langle i \rangle)$
- Output the n bits $F_s(\text{IV} \,\|\, \langle i \rangle)$ and the new state $(s, \text{IV}, \langle i+1 \rangle)$

$\langle i \rangle$ = Binary encoding of i using $n/4$ bits

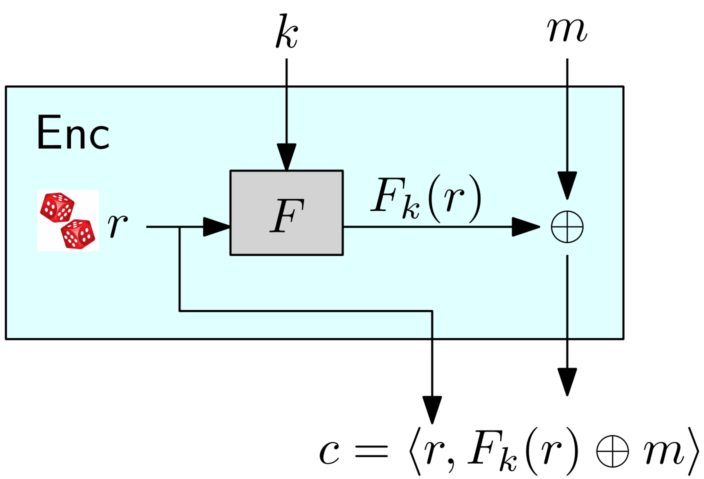
Block Ciphers: Modes of Operation

- We already have seen how to encrypt a message using a stream cipher.
- We have also seen how to encrypt a message using a block cipher (i.e., a pseudorandom permutation*)

*actually, a PRF suffices

Block Ciphers: Modes of Operation

- We already have seen how to encrypt a message using a stream cipher.
- We have also seen how to encrypt a message using a block cipher (i.e., a pseudorandom permutation*)

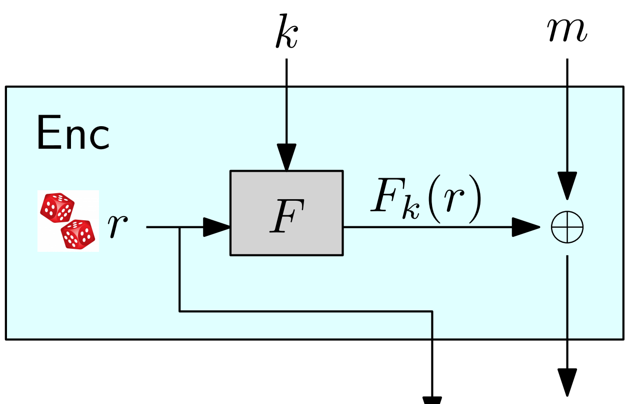


*actually, a PRF suffices

Block Ciphers: Modes of Operation

- We already have seen how to encrypt a message using a stream cipher.

- We have also seen how to encrypt a message using a block cipher (i.e., a pseudorandom permutation*)



$$c = \langle r, F_k(r) \oplus m \rangle$$

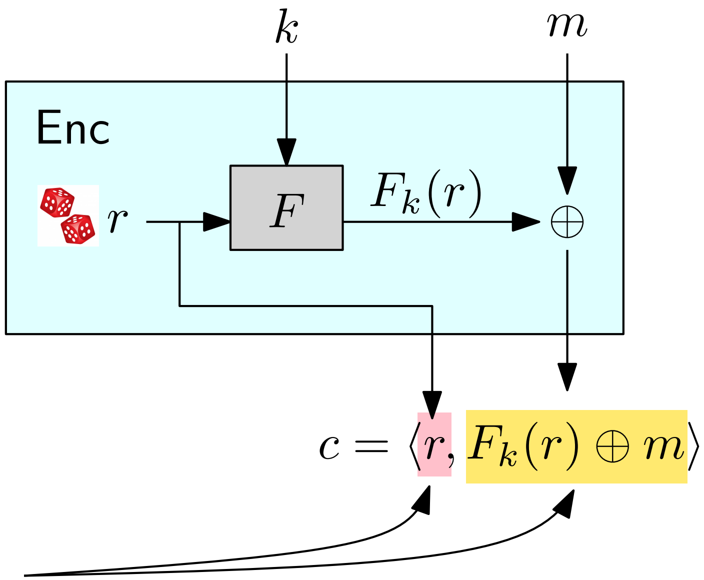
- The ciphertext is (at least) twice as long as the plaintext

*actually, a PRF suffices

Block Ciphers: Modes of Operation

- We already have seen how to encrypt a message using a stream cipher.

- We have also seen how to encrypt a message using a block cipher (i.e., a pseudorandom permutation*)



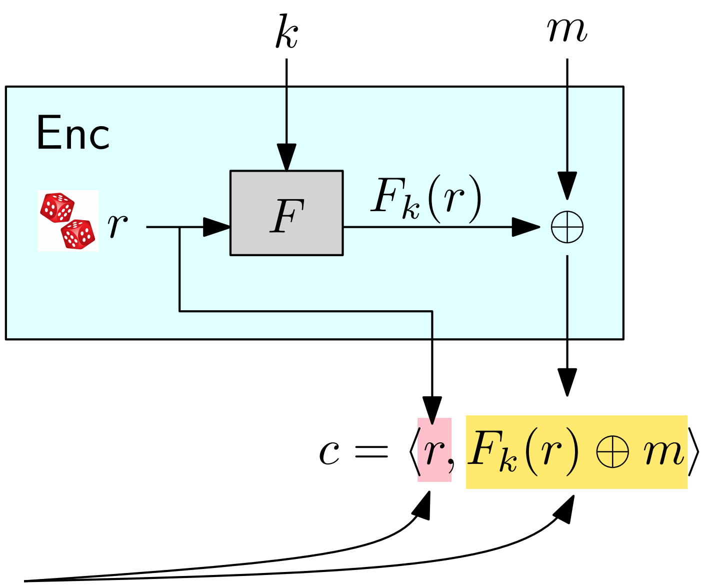
- The ciphertext is (at least) twice as long as the plaintext

- Can we do better?

*actually, a PRF suffices

Block Ciphers: Modes of Operation

- We already have seen how to encrypt a message using a stream cipher.
- We have also seen how to encrypt a message using a block cipher (i.e., a pseudorandom permutation*)



- The ciphertext is (at least) twice as long as the plaintext
- Can we do better? Several options (modes of operations)

*actually, a PRF suffices

Electronic Code Book (ECB) mode

First idea:

- Encrypt each block of the message independently

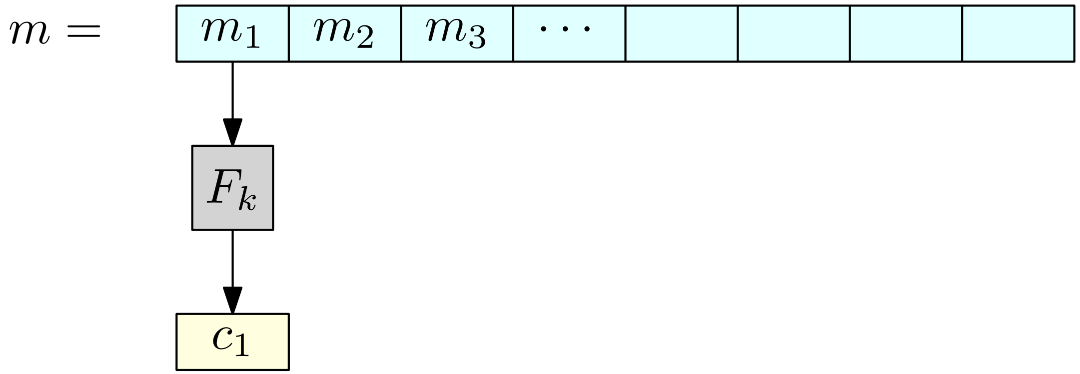
$m =$

m_1	m_2	m_3	$\cdots$				

Electronic Code Book (ECB) mode

First idea:

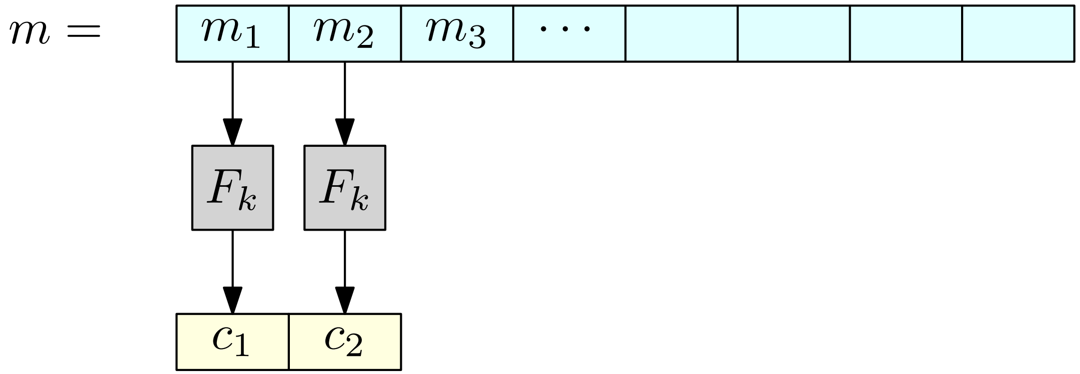
- Encrypt each block of the message independently



Electronic Code Book (ECB) mode

First idea:

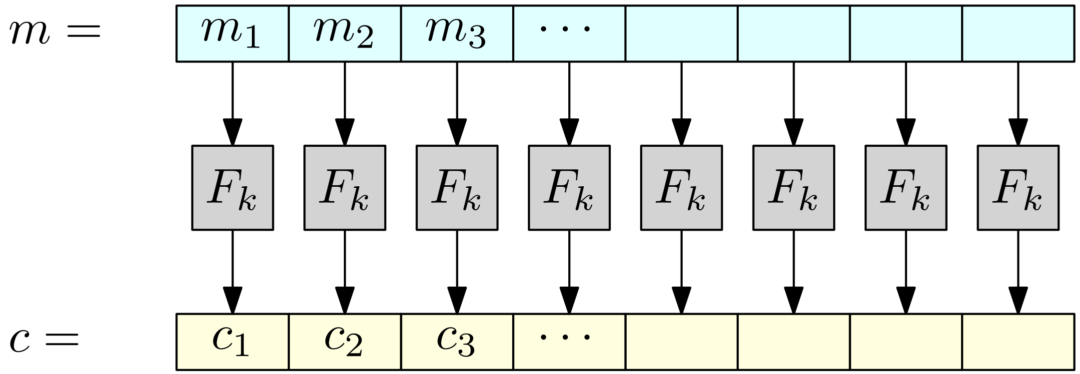
- Encrypt each block of the message independently



Electronic Code Book (ECB) mode

First idea:

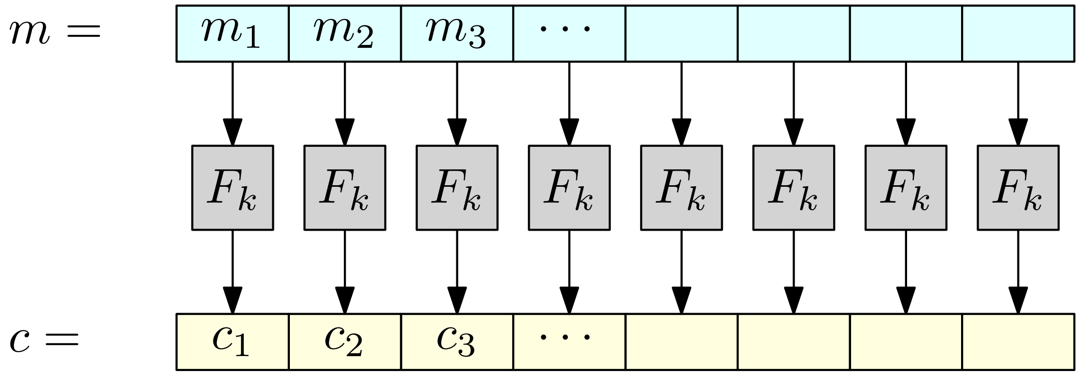
- Encrypt each block of the message independently



Electronic Code Book (ECB) mode

First idea:

- Encrypt each block of the message independently

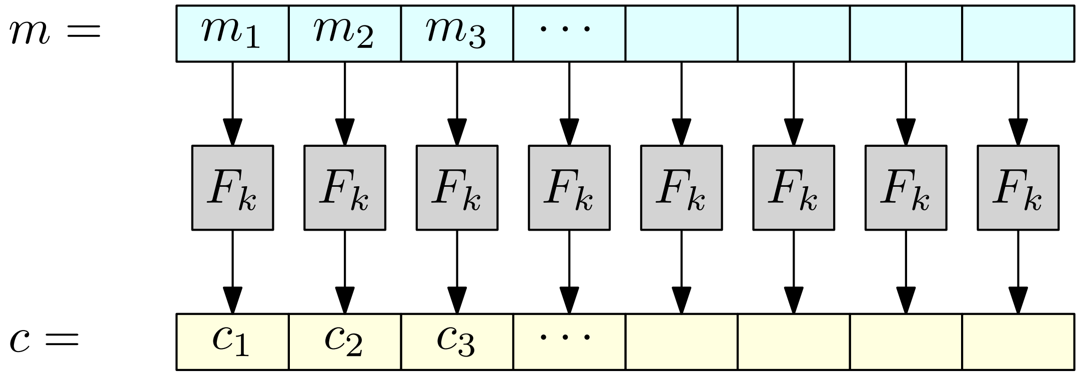


Encrypting: $c_i = F_k(m_i)$ **Decrypting:** $m_i = F_k^{-1}(c_i)$

Electronic Code Book (ECB) mode

First idea:

- Encrypt each block of the message independently



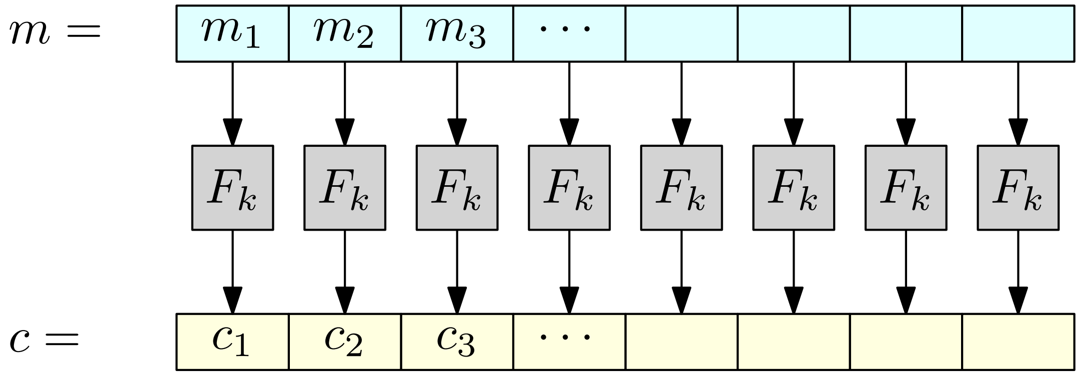
Encrypting: $c_i = F_k(m_i)$ **Decrypting:** $m_i = F_k^{-1}(c_i)$

- No ciphertext expansion!

Electronic Code Book (ECB) mode

First idea:

- Encrypt each block of the message independently



Encrypting: $c_i = F_k(m_i)$ **Decrypting:** $m_i = F_k^{-1}(c_i)$

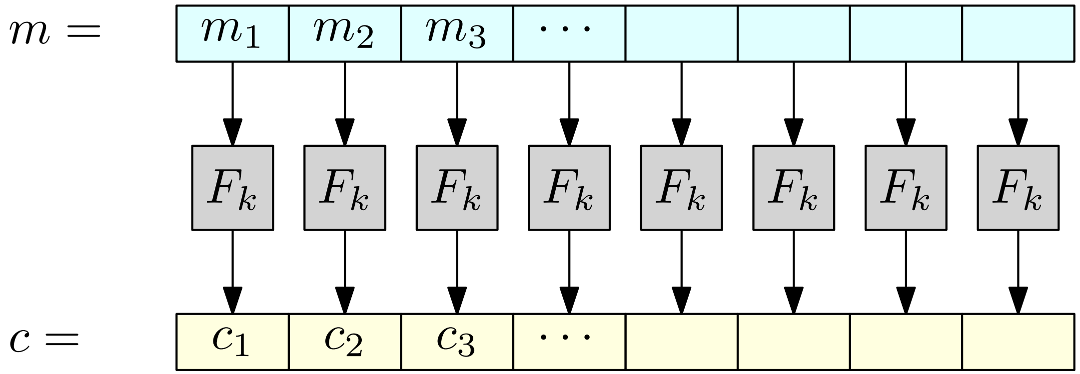
- No ciphertext expansion!

- Is it CPA-secure?

Electronic Code Book (ECB) mode

First idea:

- Encrypt each block of the message independently



Encrypting: $c_i = F_k(m_i)$ **Decrypting:** $m_i = F_k^{-1}(c_i)$

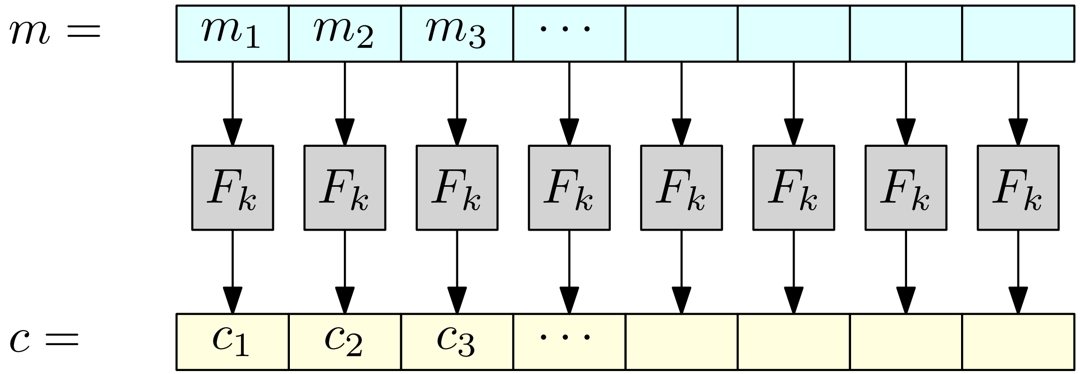
- No ciphertext expansion!

- Is it CPA-secure? No! Encryption is deterministic!

Electronic Code Book (ECB) mode

First idea:

- Encrypt each block of the message independently



Encrypting: $c_i = F_k(m_i)$ **Decrypting:** $m_i = F_k^{-1}(c_i)$

- No ciphertext expansion!

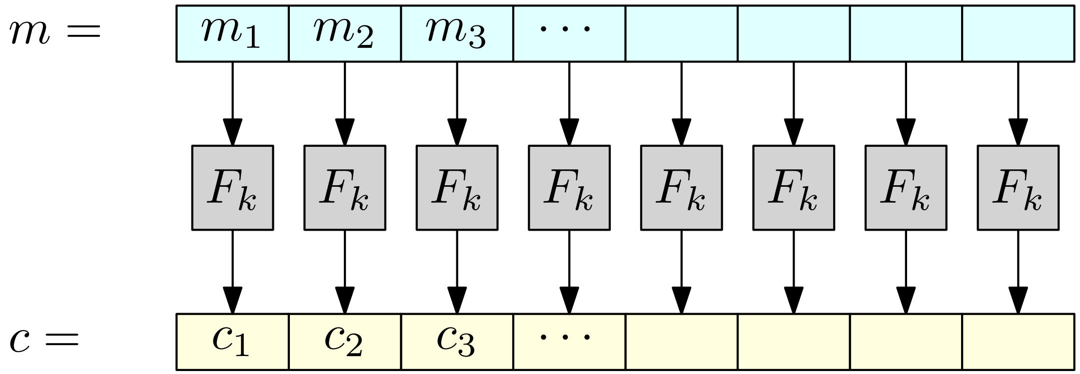
- Is it CPA-secure? No! Encryption is deterministic!

- Is it EAV-secure?

Electronic Code Book (ECB) mode

First idea:

- Encrypt each block of the message independently



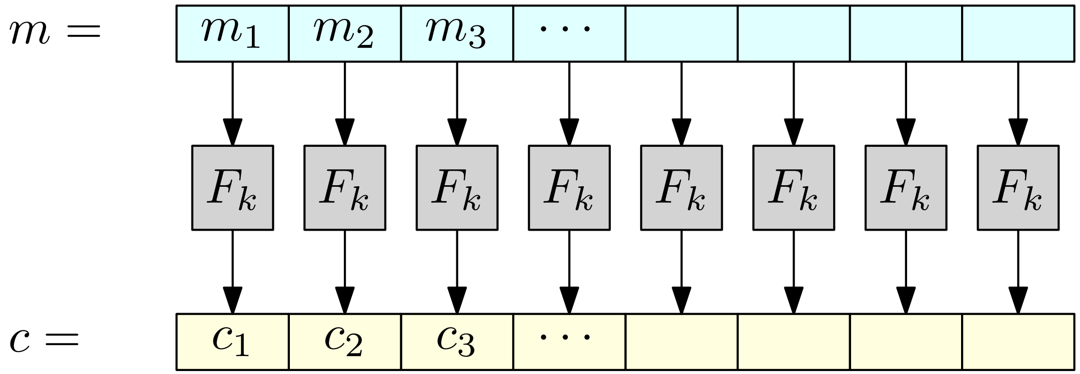
Encrypting: $c_i = F_k(m_i)$ **Decrypting:** $m_i = F_k^{-1}(c_i)$

- No ciphertext expansion!
- Is it CPA-secure? No! Encryption is deterministic!
- Is it EAV-secure? [Demo]

Electronic Code Book (ECB) mode

First idea:

- Encrypt each block of the message independently



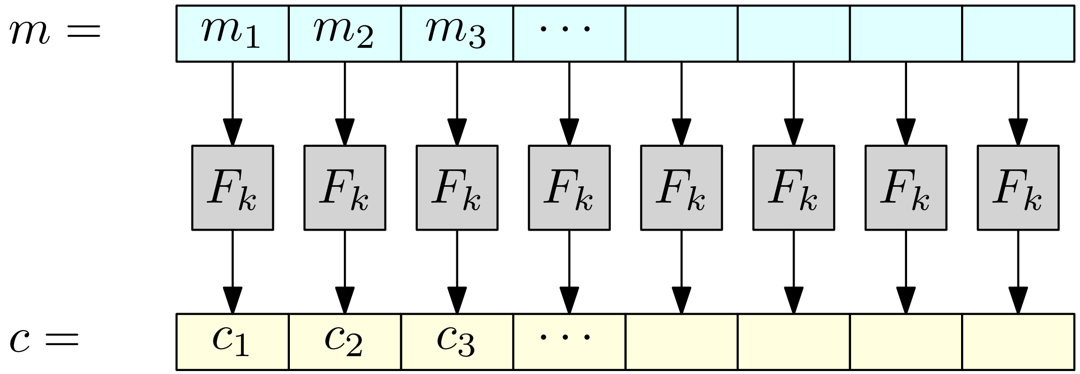
Encrypting: $c_i = F_k(m_i)$ **Decrypting:** $m_i = F_k^{-1}(c_i)$

- No ciphertext expansion!
- Is it CPA-secure? No! Encryption is deterministic!
- Is it EAV-secure? [Demo] No! It's just a fancy substitution cipher!
  (Frequency analysis)

Electronic Code Book (ECB) mode

First idea:

- Encrypt each block of the message independently



Encrypting: $c_i = F_k(m_i)$ **Decrypting:** $m_i = F_k^{-1}(c_i)$

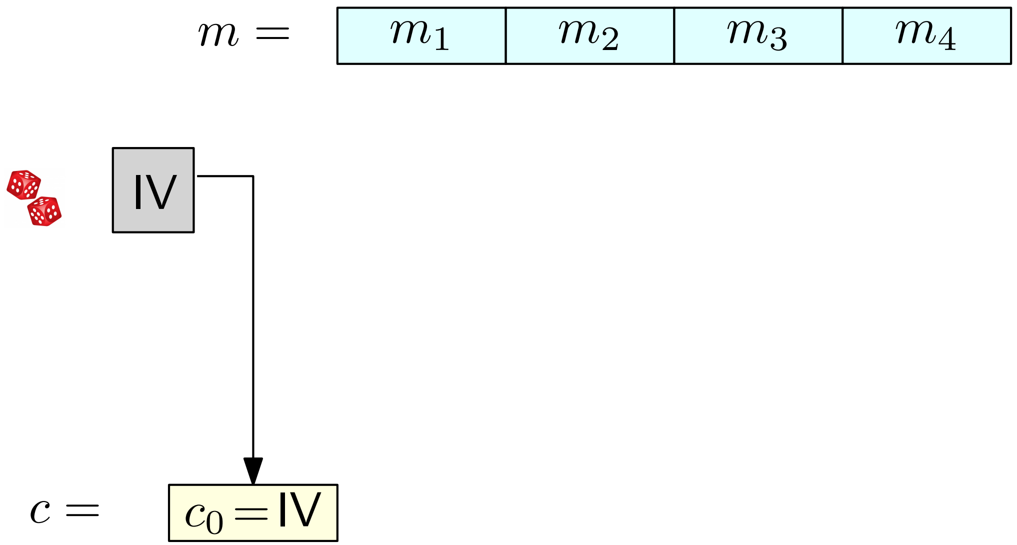
Never use ECB!

- No ciphertext expansion!

- Is it CPA-secure? No! Encryption is deterministic!

- Is it EAV-secure? [Demo] No! It's just a fancy substitution cipher! (Frequency analysis)

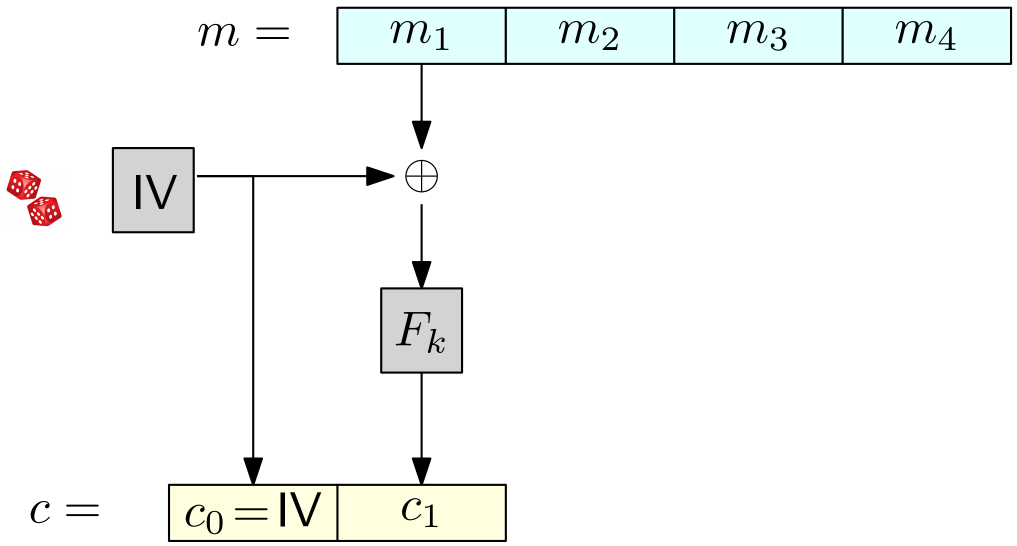
Cipher Block Chaining (CBC) mode



Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext

Cipher Block Chaining (CBC) mode

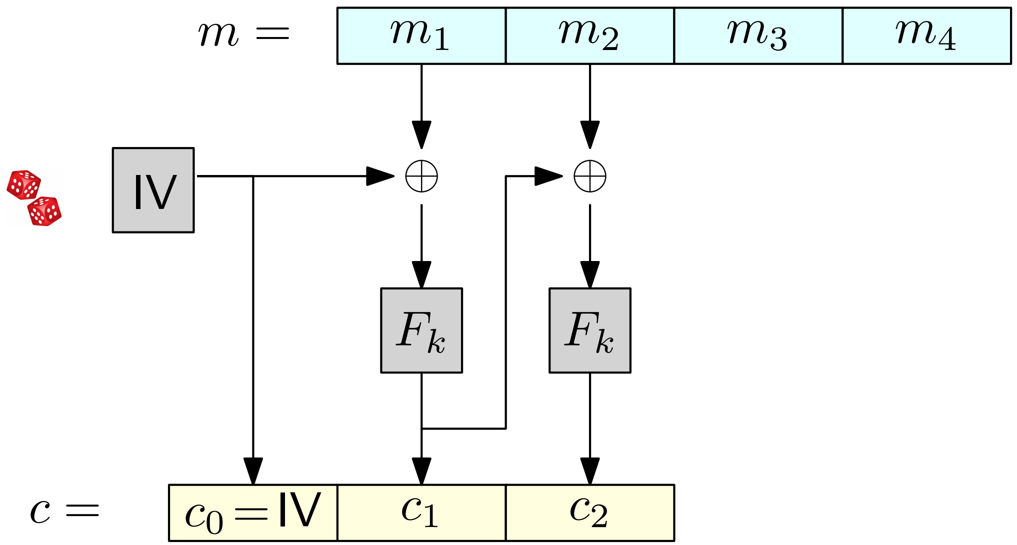


Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext
- Each block m_i of the message is XORed with the previous ciphertext block before applying F_k

$$c_i = F_k(c_{i-1} \oplus m_i)$$

Cipher Block Chaining (CBC) mode

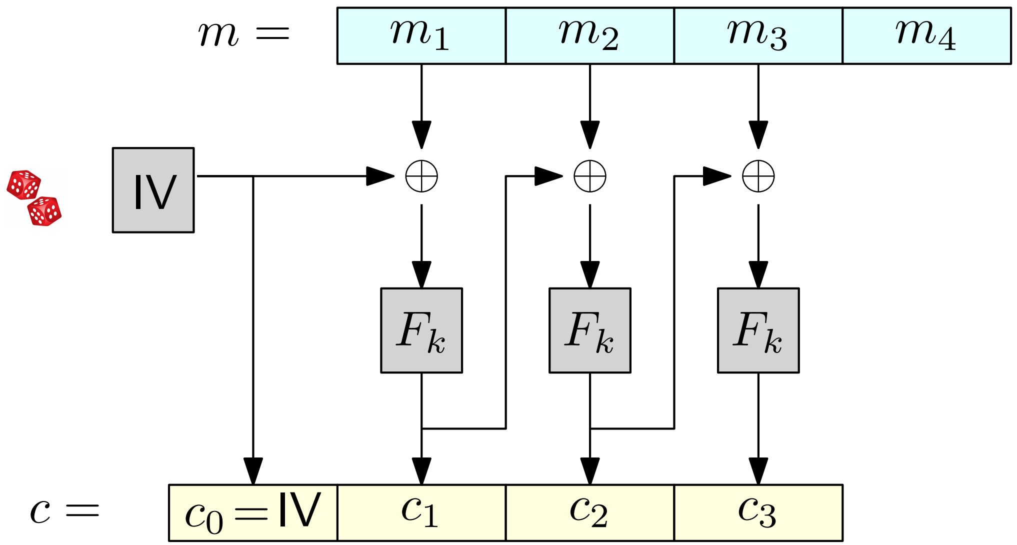


Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext
- Each block m_i of the message is XORed with the previous ciphertext block before applying F_k

$$c_i = F_k(c_{i-1} \oplus m_i)$$

Cipher Block Chaining (CBC) mode

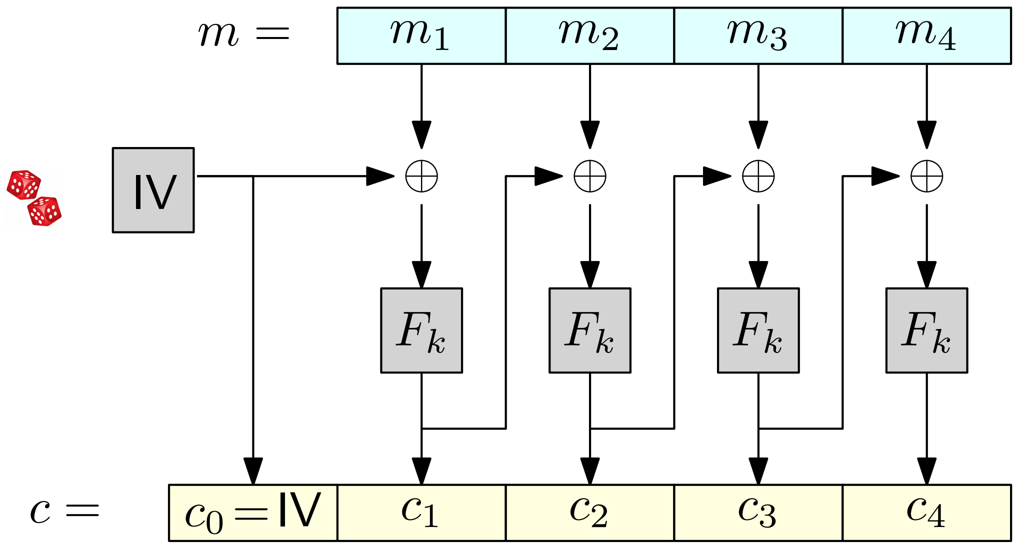


Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext
- Each block m_i of the message is XORed with the previous ciphertext block before applying F_k

$$c_i = F_k(c_{i-1} \oplus m_i)$$

Cipher Block Chaining (CBC) mode

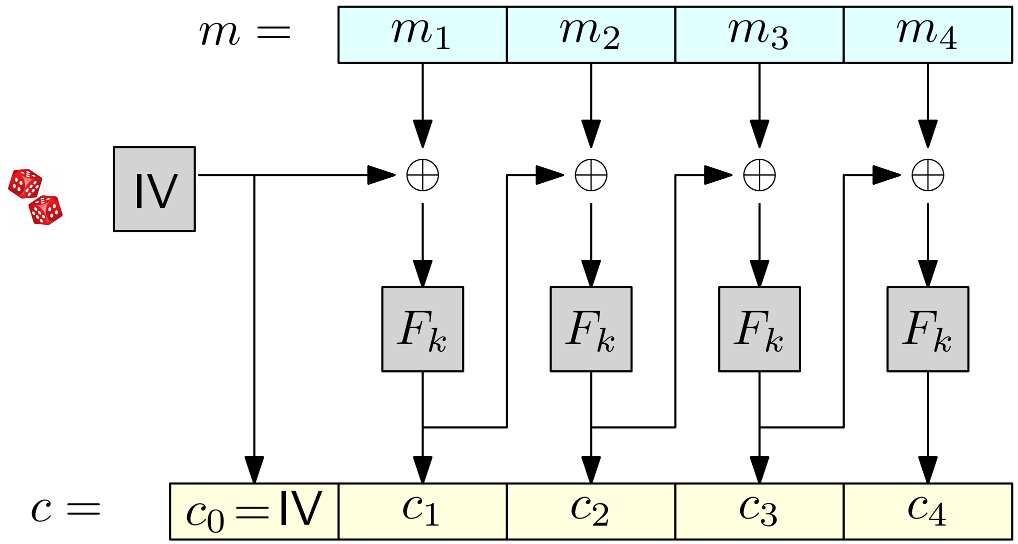


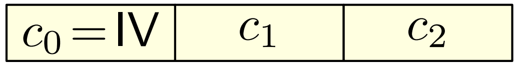
Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext
- Each block m_i of the message is XORed with the previous ciphertext block before applying F_k

$$c_i = F_k(c_{i-1} \oplus m_i)$$

Cipher Block Chaining (CBC) mode: Decrypting

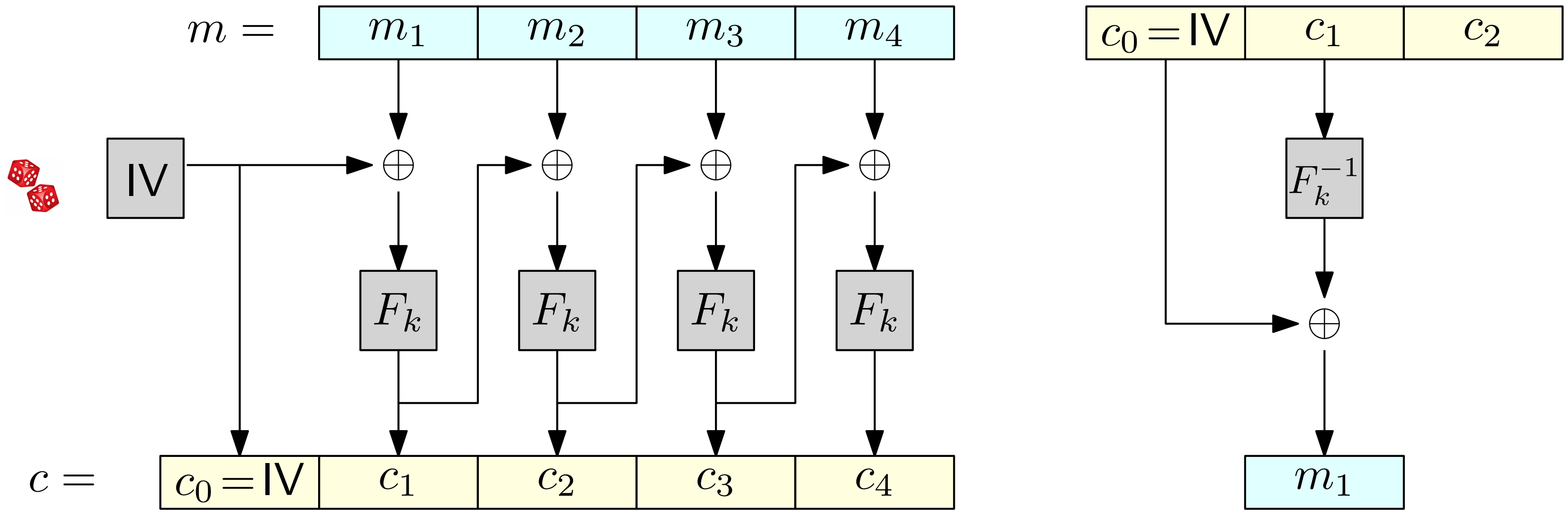




Decrypting:

- To decrypt m_i we need c_{i-1}

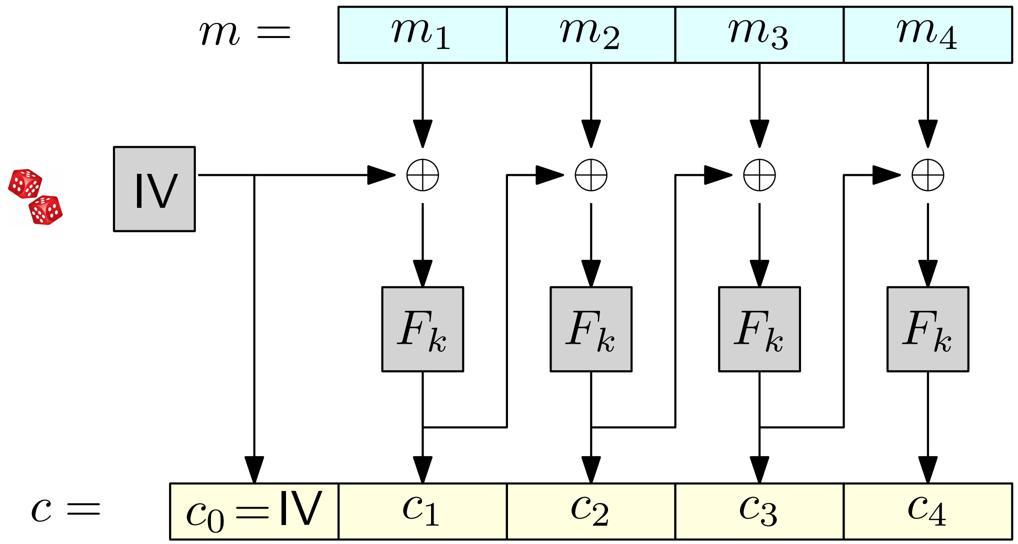
Cipher Block Chaining (CBC) mode: Decrypting

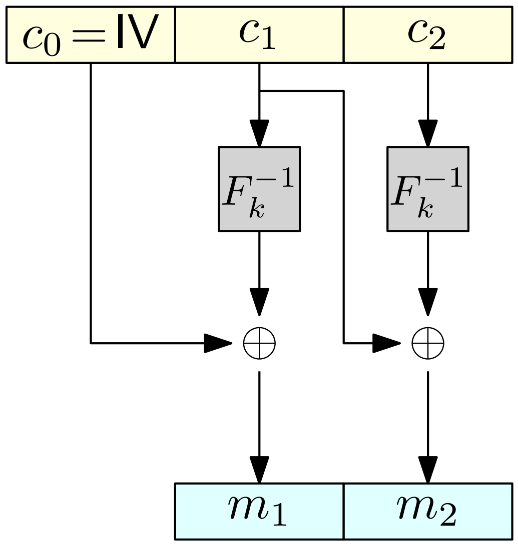


Decrypting:

- To decrypt m_i we need c_{i-1}
- $m_i = F_k^{-1}(c_i) \oplus c_{i-1}$

Cipher Block Chaining (CBC) mode: Decrypting

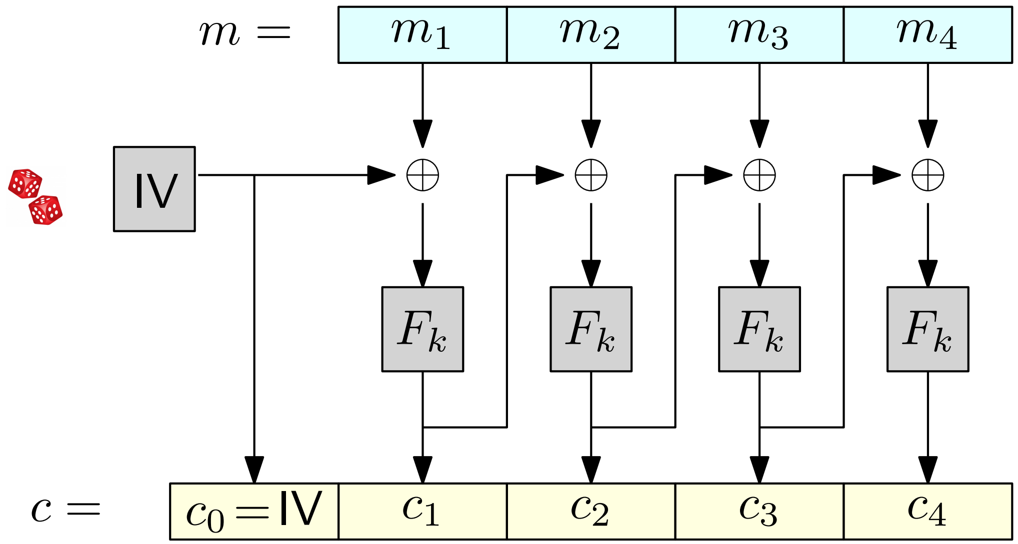


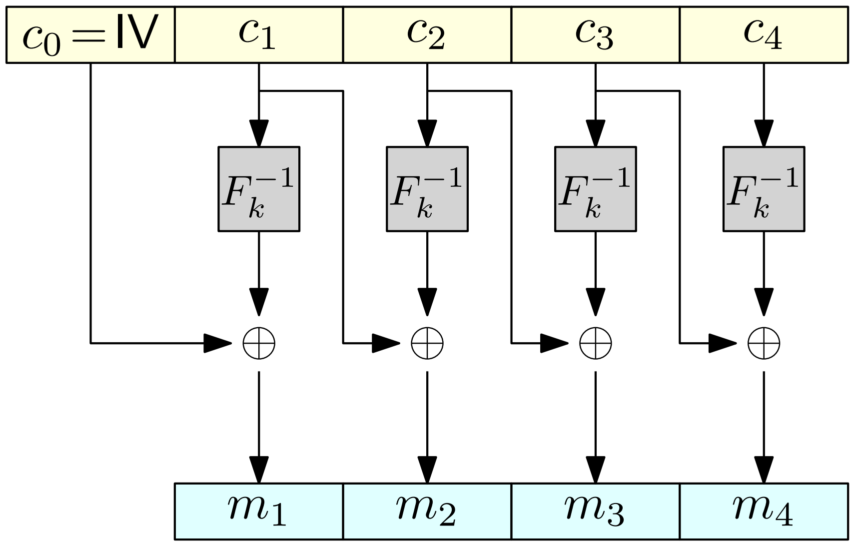


Decrypting:

- To decrypt m_i we need c_{i-1}
- $m_i = F_k^{-1}(c_i) \oplus c_{i-1}$

Cipher Block Chaining (CBC) mode: Decrypting

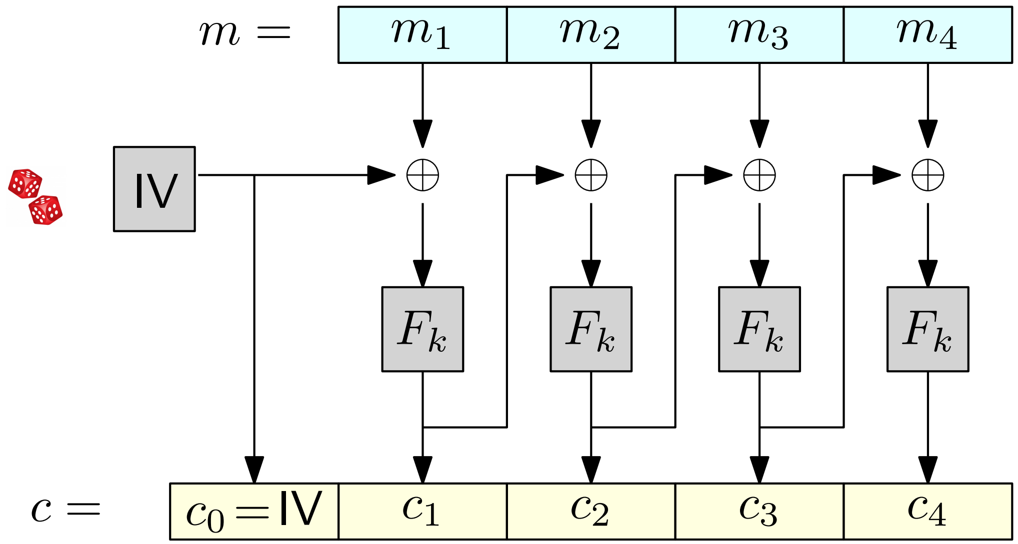


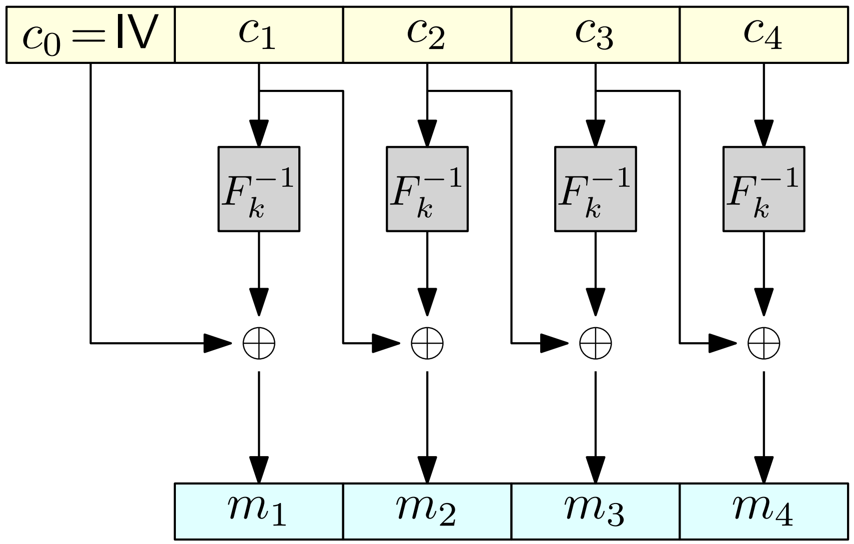


Decrypting:

- To decrypt m_i we need c_{i-1}
- $m_i = F_k^{-1}(c_i) \oplus c_{i-1}$

Cipher Block Chaining (CBC) mode: Decrypting



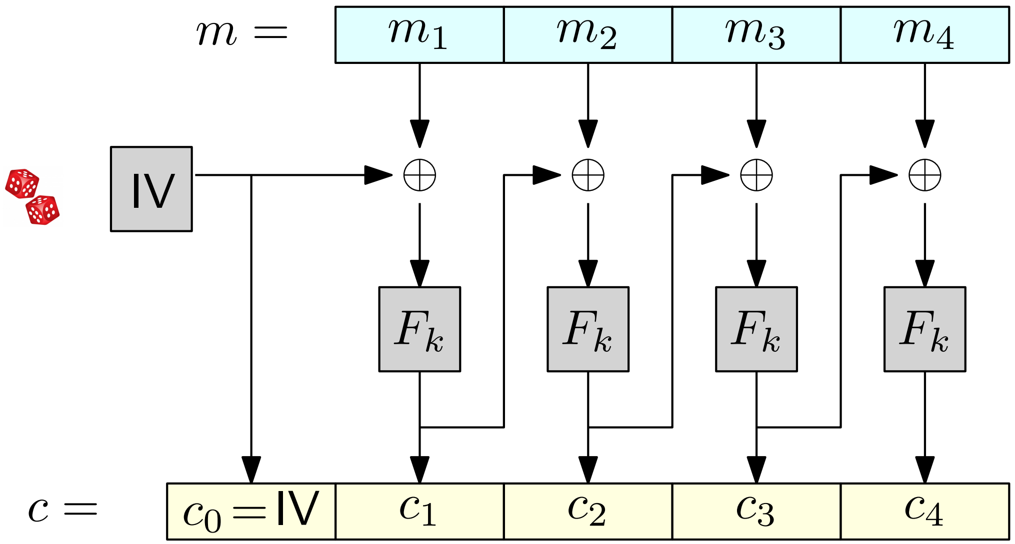


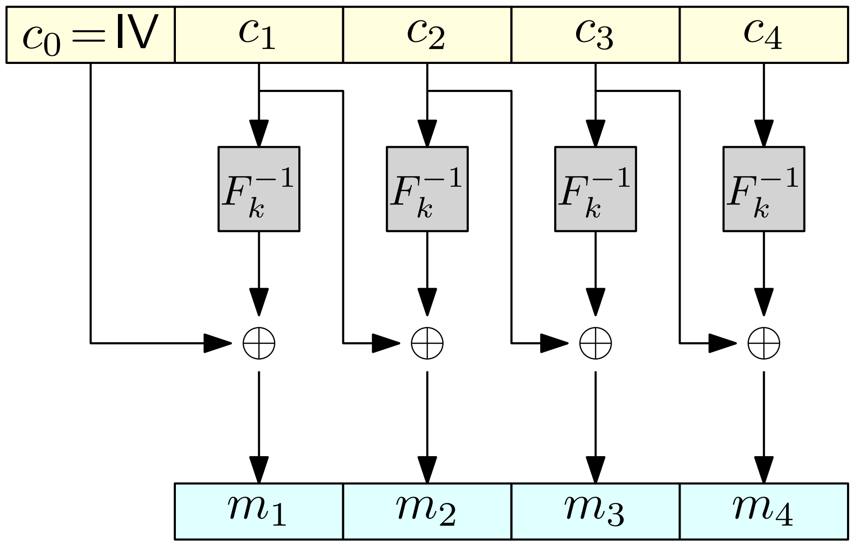
Decrypting:

- To decrypt m_i we need c_{i-1}
- $m_i = F_k^{-1}(c_i) \oplus c_{i-1}$

Drawback: Encryption must be done sequentially

Cipher Block Chaining (CBC) mode: Decrypting





Decrypting:

- To decrypt m_i we need c_{i-1}
- $m_i = F_k^{-1}(c_i) \oplus c_{i-1}$

Drawback: Encryption must be done sequentially (but decryption can be done in parallel)

Cipher Block Chaining (CBC) mode

Is CBC mode CPA secure?

Cipher Block Chaining (CBC) mode

Is CBC mode CPA secure? Yes!*

Cipher Block Chaining (CBC) mode

Is CBC mode CPA secure? Yes!*

Theorem: If F is a pseudorandom permutation, then CBC mode is CPA-secure.

Cipher Block Chaining (CBC) mode

Is CBC mode CPA secure? Yes!*

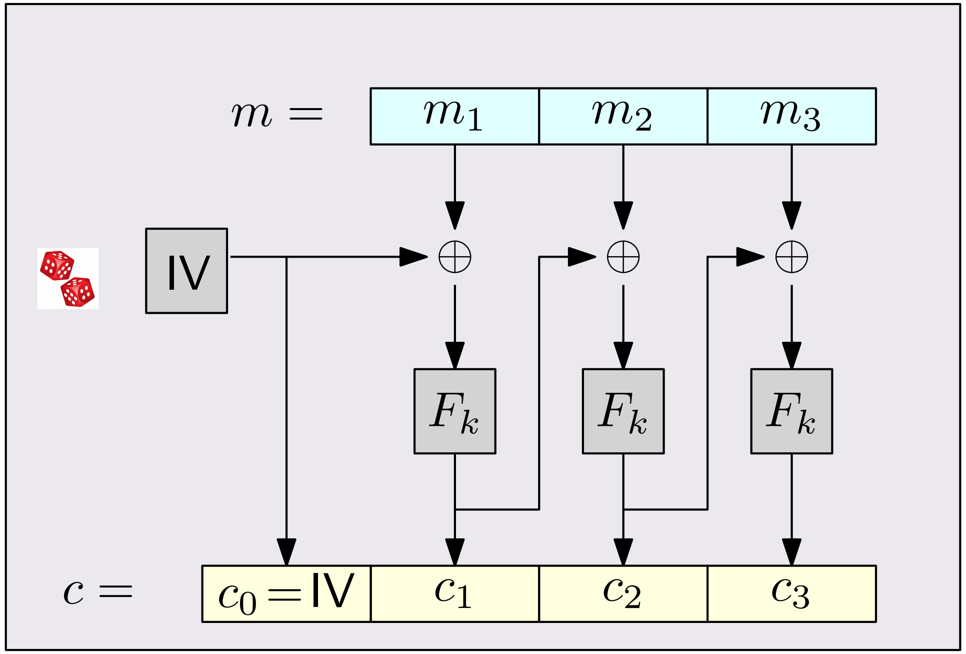
Theorem: If F is a pseudorandom permutation, then CBC mode is CPA-secure.

*But, depending on the implementation, it might be vulnerable to some subtle attacks
(not really a fault of the encryption scheme, but something to be aware of)

Chained CBC mode

There is a stateful variant of CBC called **chained CBC** that handles multiple messages as follows:

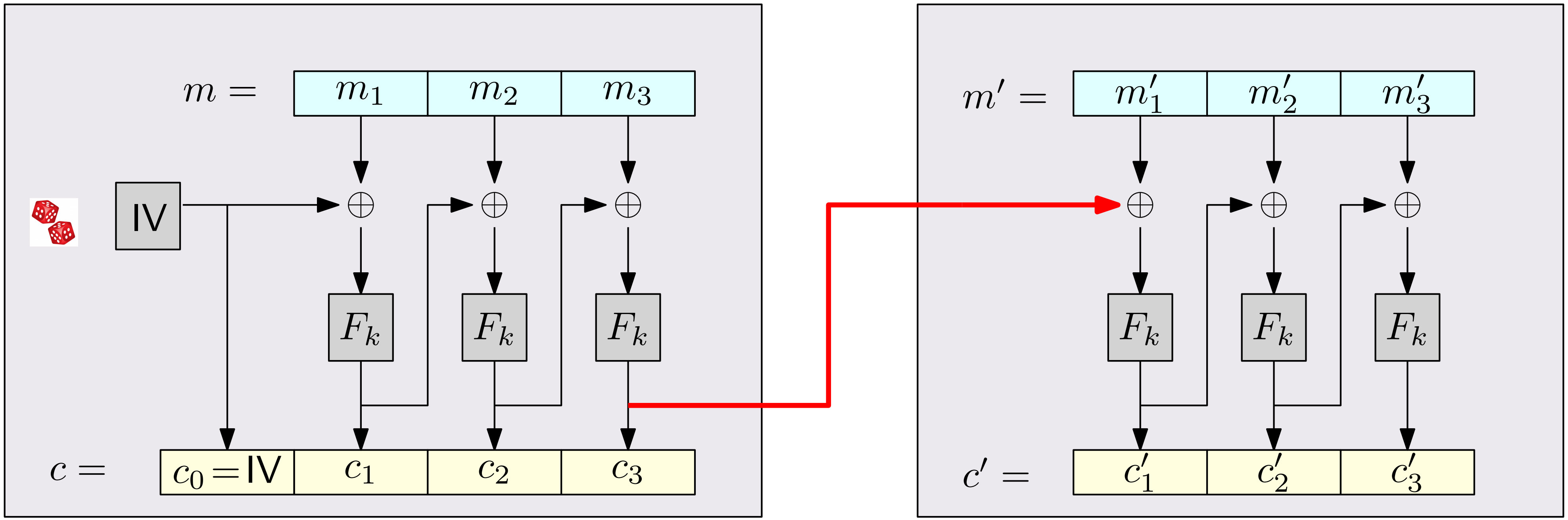
- When the first message is encrypted a random IV is chosen (like in CBC mode)



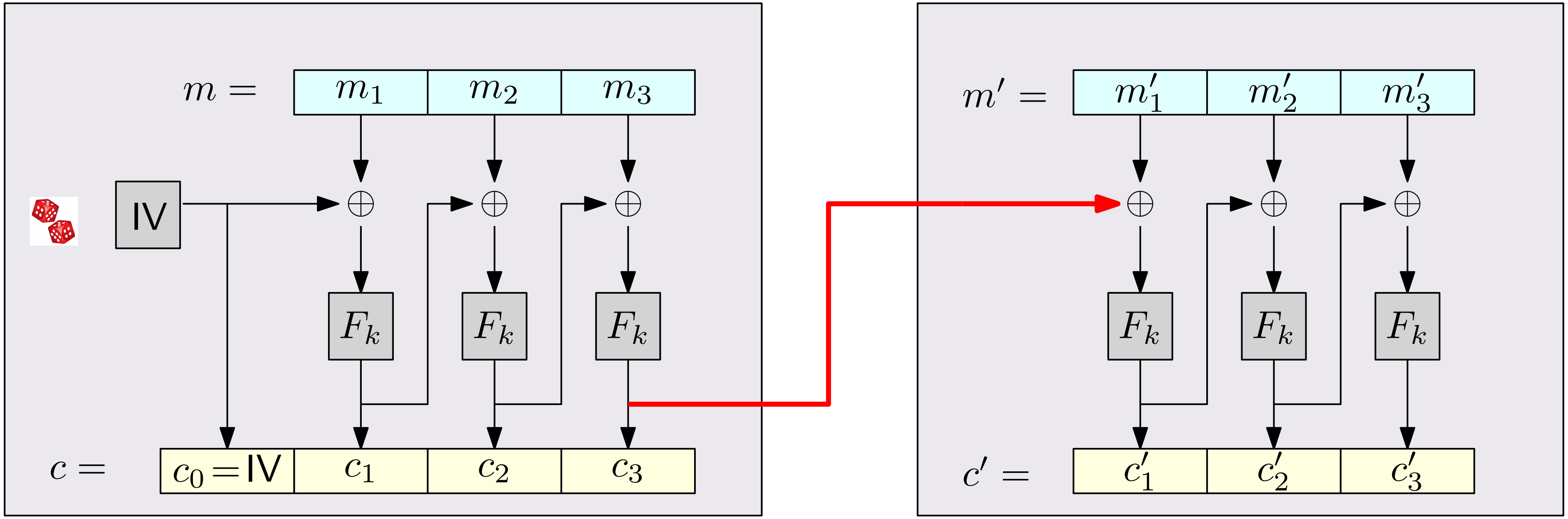
Chained CBC mode

There is a stateful variant of CBC called **chained CBC** that handles multiple messages as follows:

- When the first message is encrypted a random IV is chosen (like in CBC mode)
- When a subsequent message needs to be encrypted, the last block of the previous ciphertext is used instead of a new IV

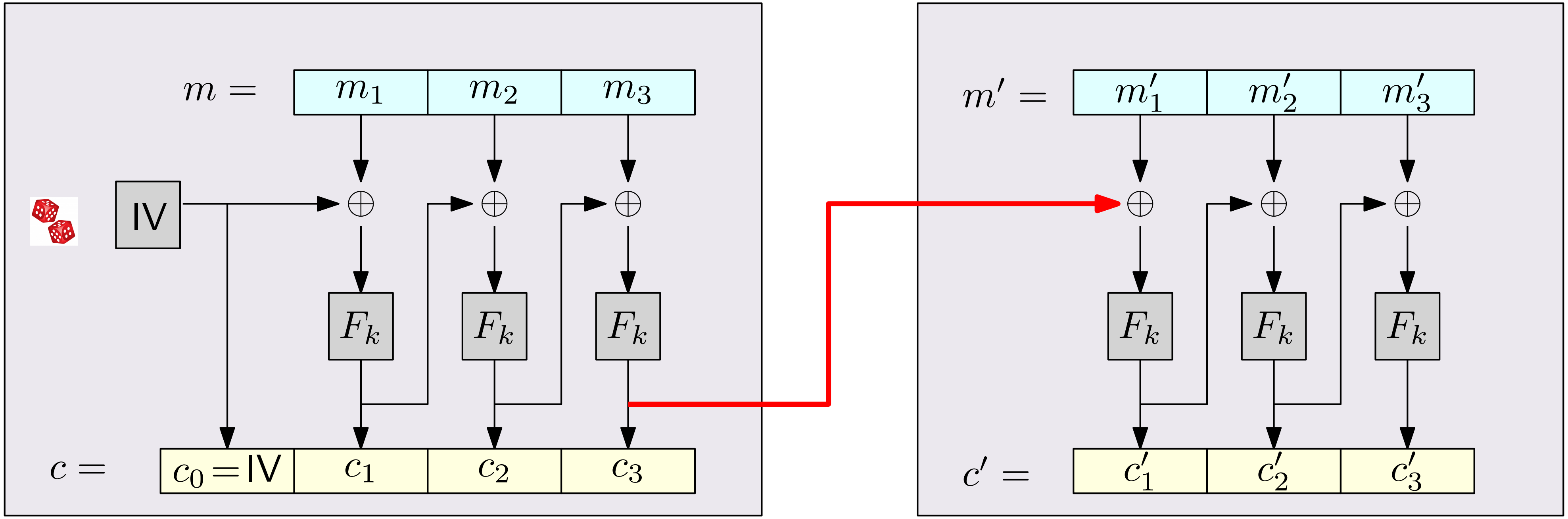


Security of Chained CBC mode



Is chained CBC mode CPA-secure?

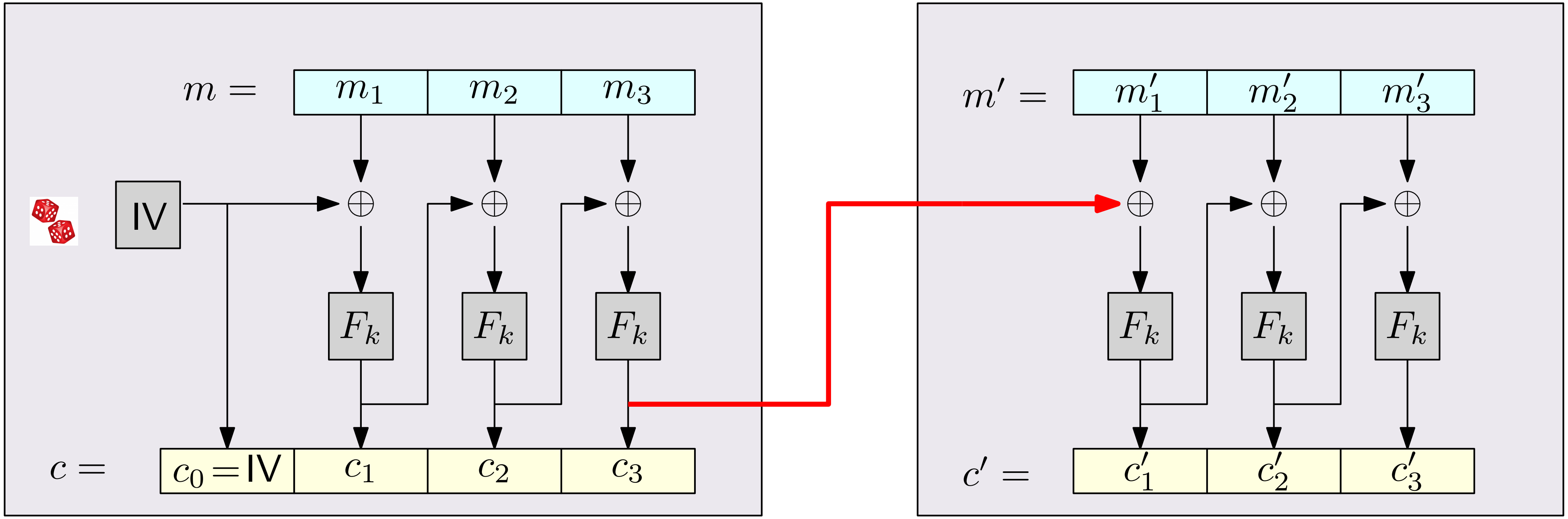
Security of Chained CBC mode



Is chained CBC mode CPA-secure?

We are just simulating CBC mode on a bigger message $m \| m'$...

Security of Chained CBC mode

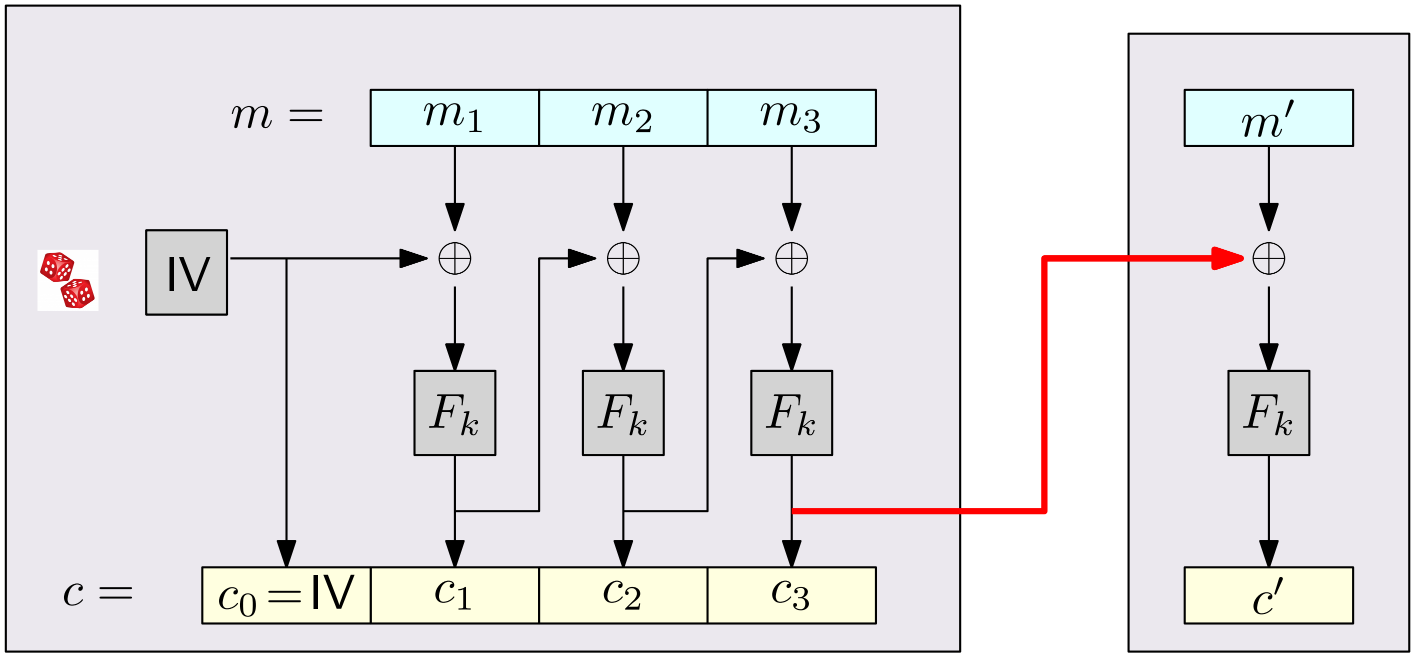


Is chained CBC mode CPA-secure?

We are just simulating CBC mode on a bigger message $m \| m'$...

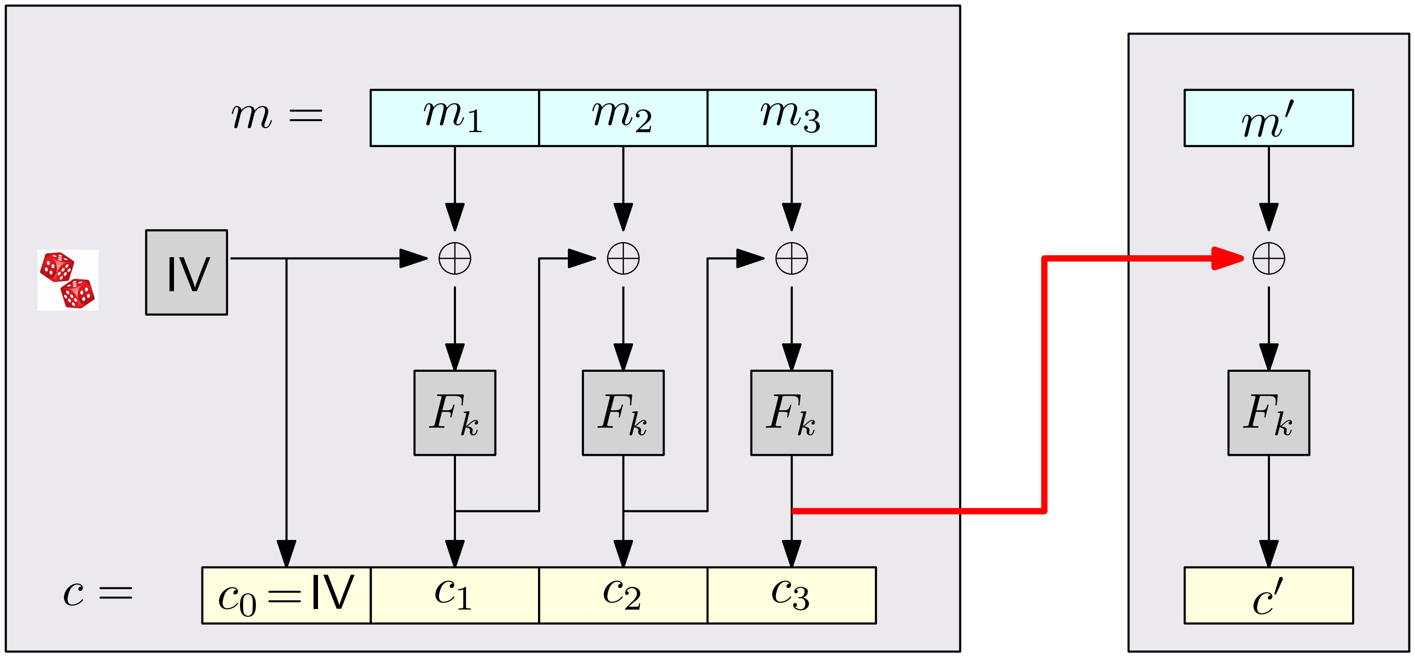
No!

Security of Chained CBC mode



Suppose that the adversary observes c and knows that m_1 is either x or y (e.g., $x =$ `ATTACK!` and $y =$ `RETREAT`)

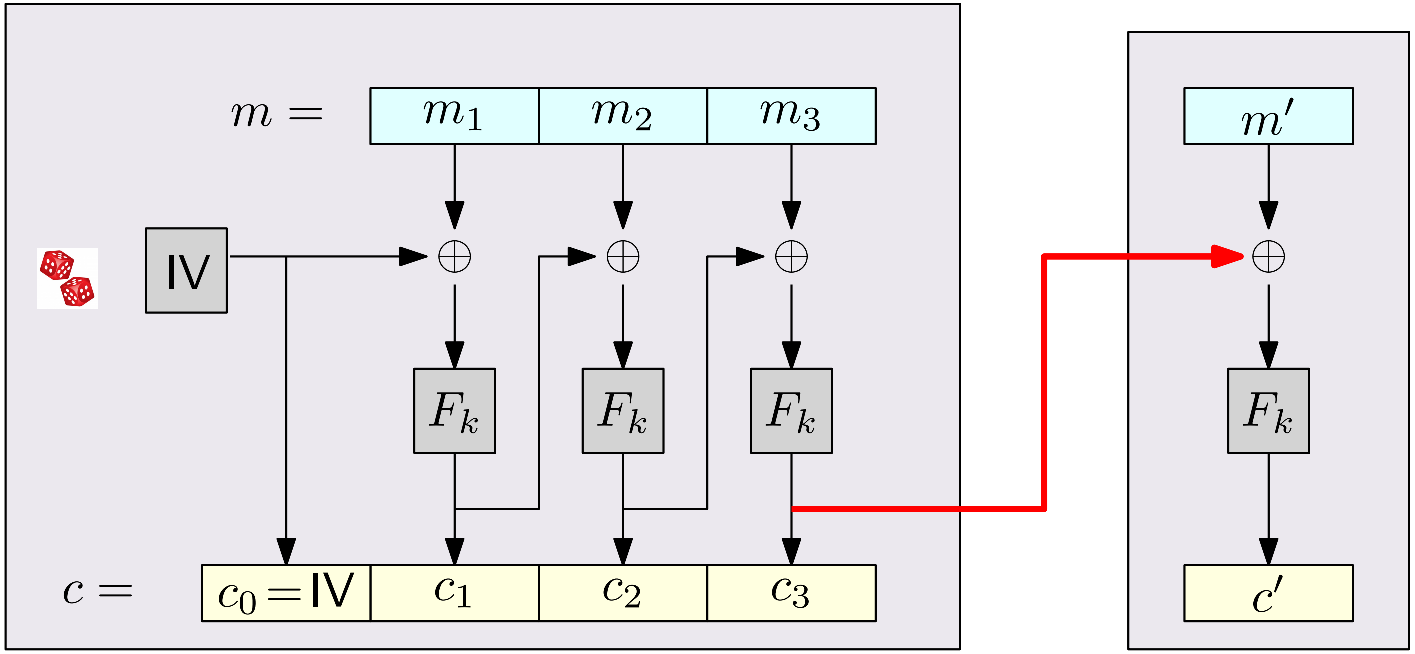
Security of Chained CBC mode



Suppose that the adversary observes c and knows that m_1 is either x or y (e.g., $x = \texttt{ATTACK!}$ and $y = \texttt{RETREAT}$)

The adversary convinces Alice to encrypt $m' = c_0 \oplus x \oplus c_3$

Security of Chained CBC mode

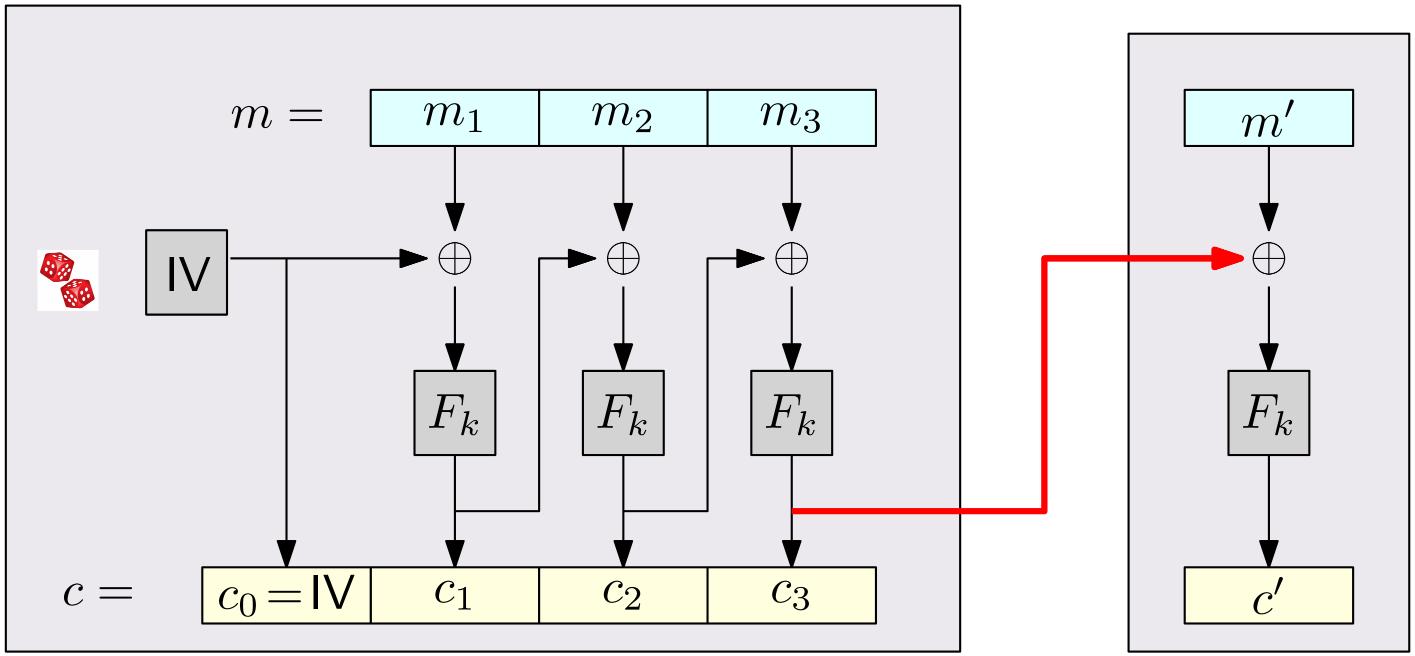


Suppose that the adversary observes c and knows that m_1 is either x or y (e.g., $x =$ `ATTACK!` and $y =$ `RETREAT`)

The adversary convinces Alice to encrypt $m' = c_0 \oplus x \oplus c_3$

If $m_1 = x$ then $c' = F_k(c_3 \oplus m')$

Security of Chained CBC mode

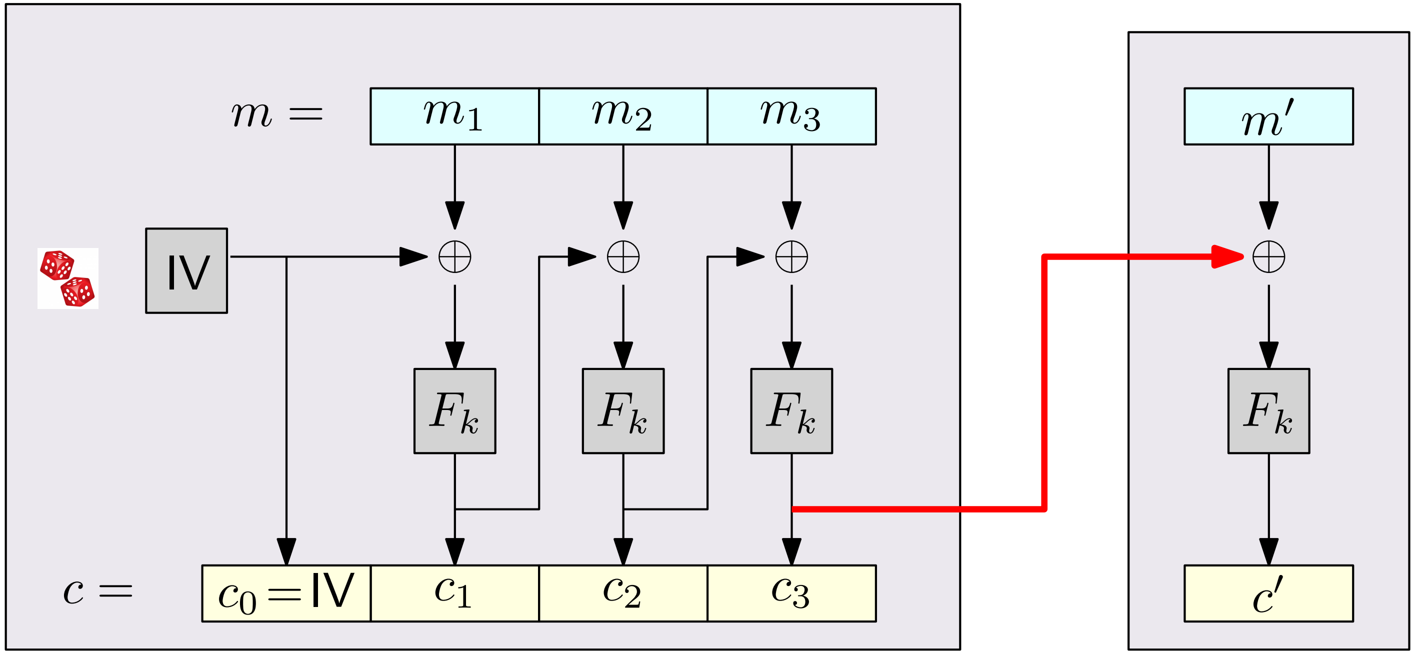


Suppose that the adversary observes c and knows that m_1 is either x or y (e.g., $x = \texttt{ATTACK!}$ and $y = \texttt{RETREAT}$)

The adversary convinces Alice to encrypt $m' = c_0 \oplus x \oplus c_3$

If $m_1 = x$ then $c' = F_k(c_3 \oplus m') = F_k(c_3 \oplus c_0 \oplus x \oplus c_3)$

Security of Chained CBC mode

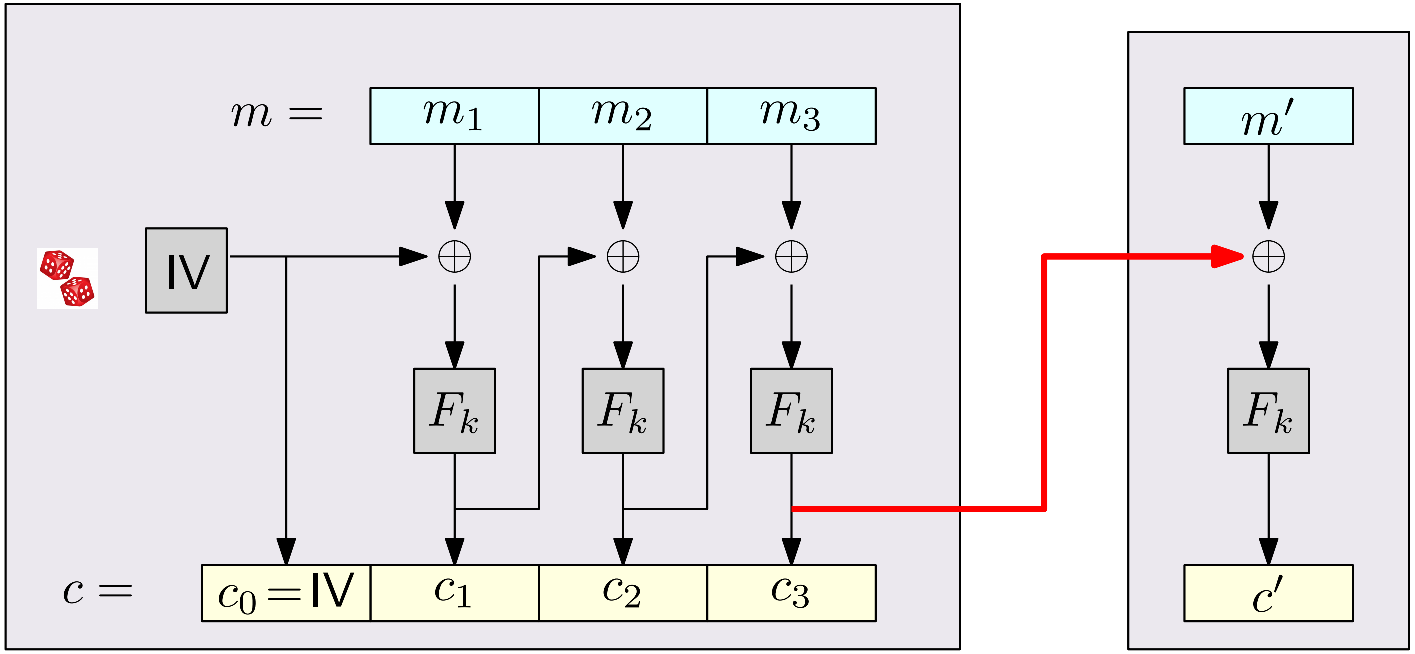


Suppose that the adversary observes c and knows that m_1 is either x or y (e.g., $x =$ `ATTACK!` and $y =$ `RETREAT`)

The adversary convinces Alice to encrypt $m' = c_0 \oplus x \oplus c_3$

If $m_1 = x$ then $c' = F_k(c_3 \oplus m') = F_k(c_3 \oplus c_0 \oplus x \oplus c_3) = F_k(c_0 \oplus x)$

Security of Chained CBC mode

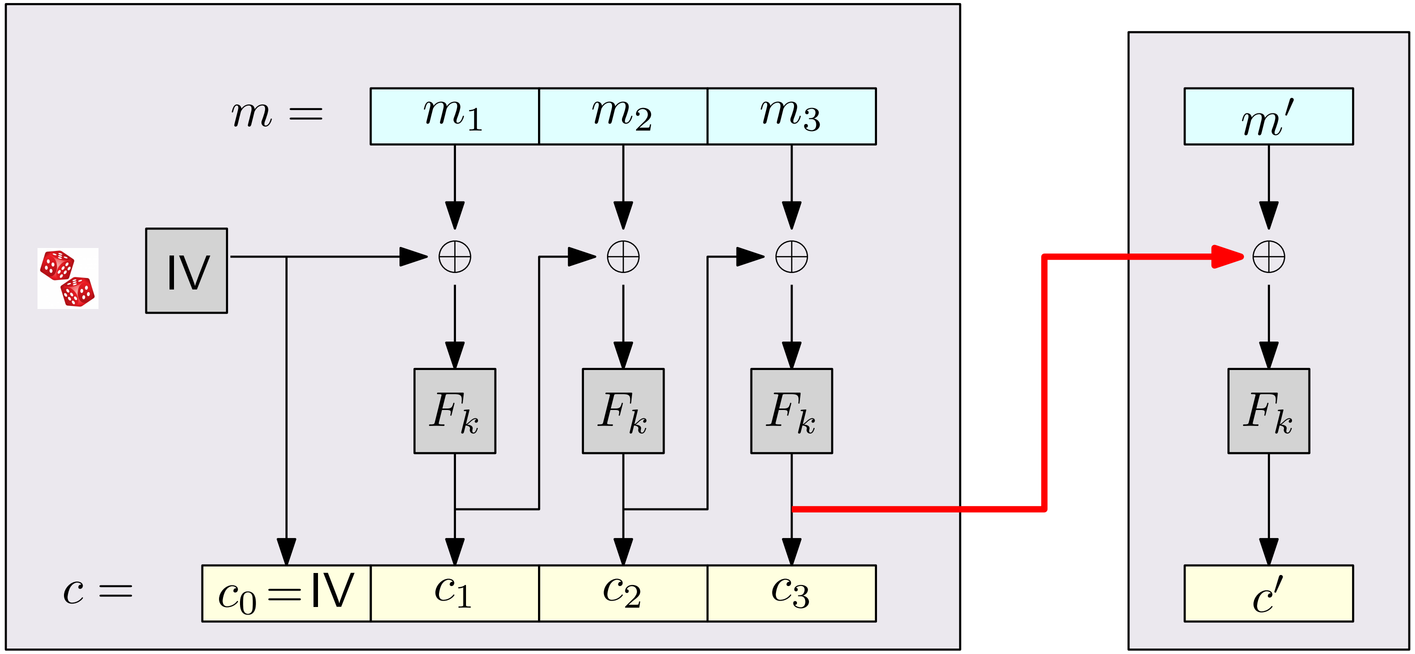


Suppose that the adversary observes c and knows that m_1 is either x or y (e.g., $x = \texttt{ATTACK!}$ and $y = \texttt{RETREAT}$)

The adversary convinces Alice to encrypt $m' = c_0 \oplus x \oplus c_3$

If $m_1 = x$ then $c' = F_k(c_3 \oplus m') = F_k(c_3 \oplus c_0 \oplus x \oplus c_3) = F_k(c_0 \oplus x) = F_k(c_0 \oplus m_1) = c_1$

Security of Chained CBC mode



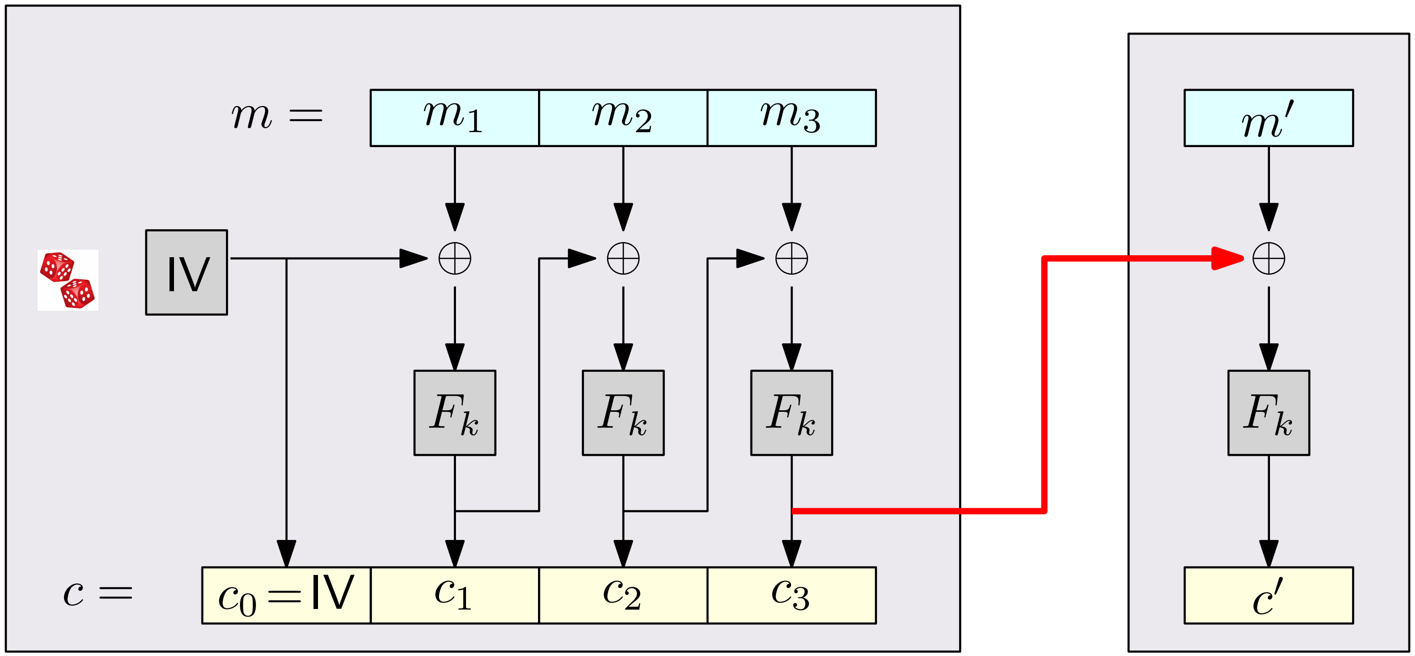
Suppose that the adversary observes c and knows that m_1 is either x or y (e.g., $x = \texttt{ATTACK!}$ and $y = \texttt{RETREAT}$)

The adversary convinces Alice to encrypt $m' = c_0 \oplus x \oplus c_3$

If $m_1 = x$ then $c' = F_k(c_3 \oplus m') = F_k(c_3 \oplus c_0 \oplus x \oplus c_3) = F_k(c_0 \oplus x) = F_k(c_0 \oplus m_1) = c_1$

If $m_1 \neq x$ then $c' = F_k(c_3 \oplus m')$

Security of Chained CBC mode



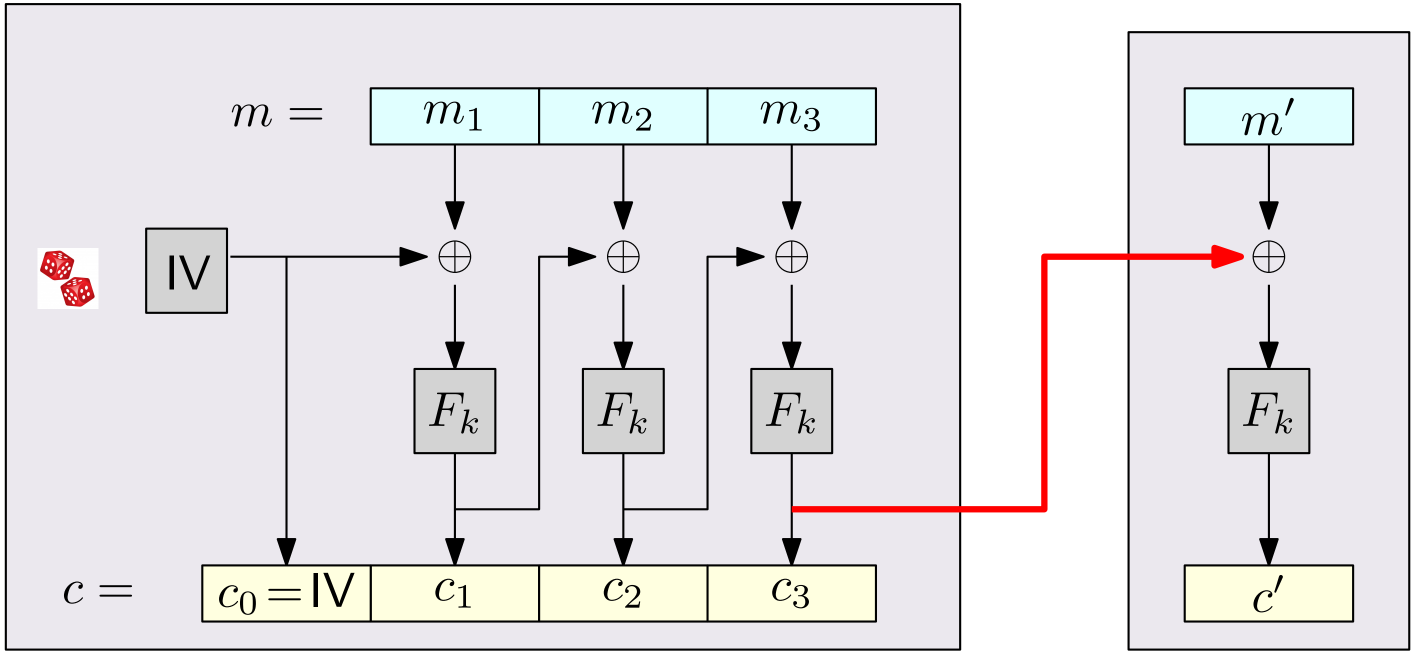
Suppose that the adversary observes c and knows that m_1 is either x or y (e.g., $x = \texttt{ATTACK!}$ and $y = \texttt{RETREAT}$)

The adversary convinces Alice to encrypt $m' = c_0 \oplus x \oplus c_3$

If $m_1 = x$ then $c' = F_k(c_3 \oplus m') \ = F_k(c_3 \oplus c_0 \oplus x \oplus c_3) \ = F_k(c_0 \oplus x) \ = F_k(c_0 \oplus m_1) = c_1$

If $m_1 \neq x$ then $c' = F_k(c_3 \oplus m') \ = F_k(c_0 \oplus x)$

Security of Chained CBC mode



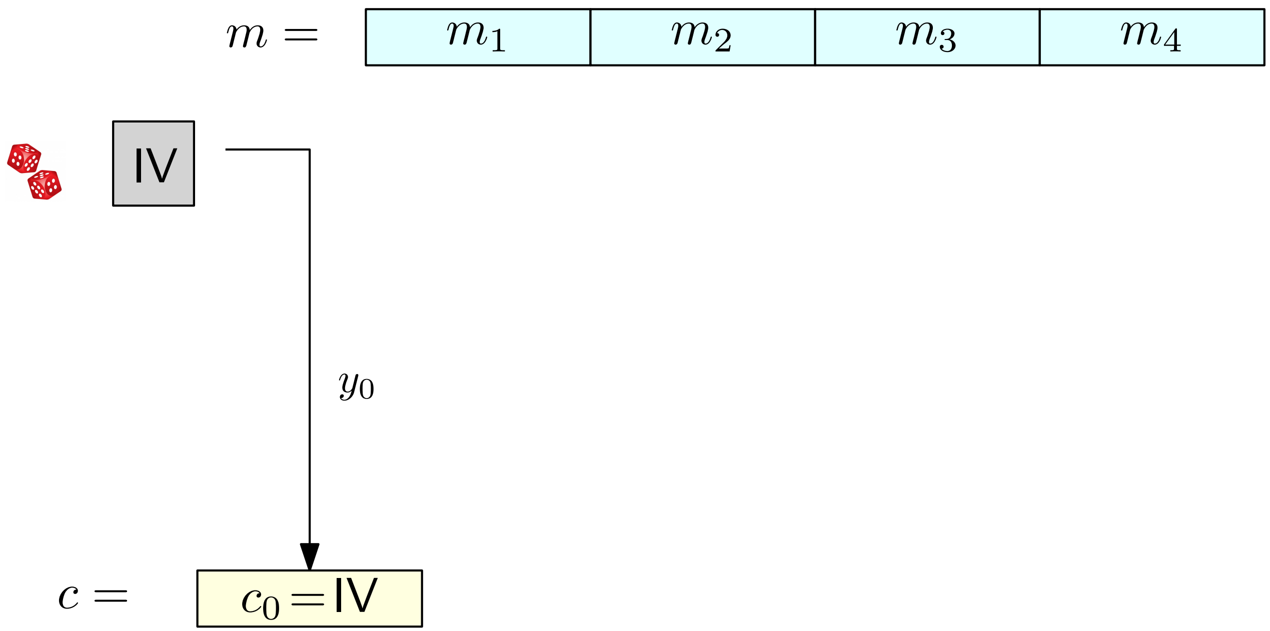
Suppose that the adversary observes c and knows that m_1 is either x or y (e.g., $x = \texttt{ATTACK!}$ and $y = \texttt{RETREAT}$)

The adversary convinces Alice to encrypt $m' = c_0 \oplus x \oplus c_3$

If $m_1 = x$ then $c' = F_k(c_3 \oplus m') = F_k(c_3 \oplus c_0 \oplus x \oplus c_3) = F_k(c_0 \oplus x) = F_k(c_0 \oplus m_1) = c_1$

If $m_1 \neq x$ then $c' = F_k(c_3 \oplus m') = F_k(c_0 \oplus x) \neq F(c_0 \oplus m_1) = c_1$

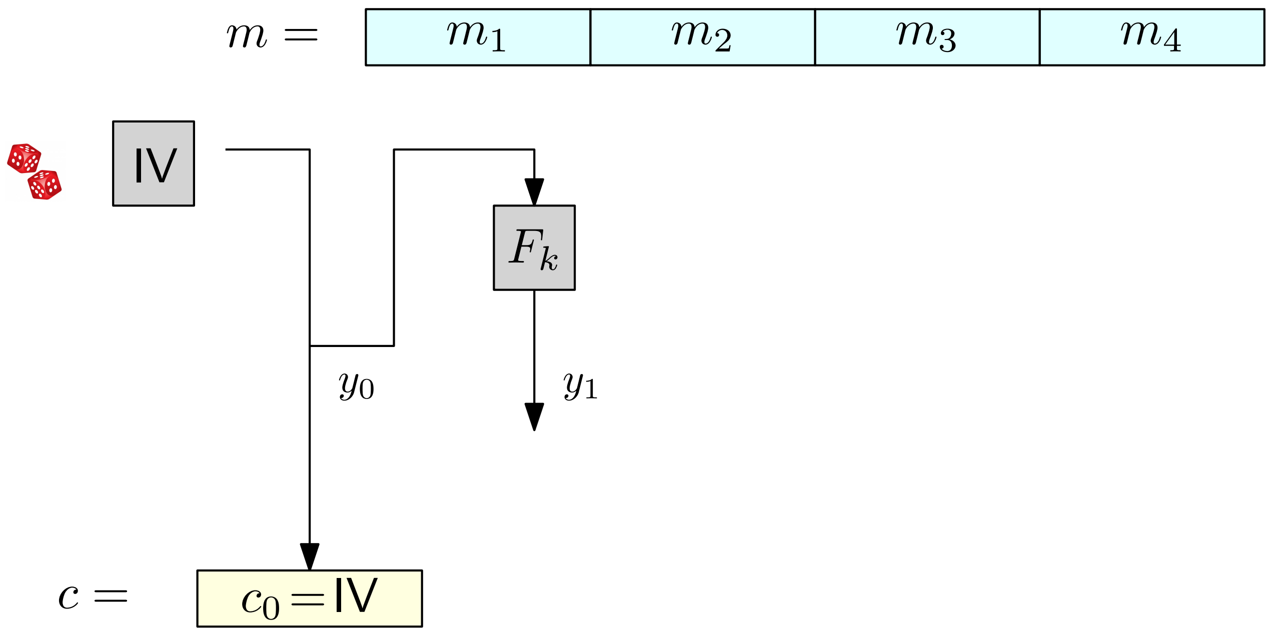
Output Feedback (OFB) mode



Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext. Let $y_0 = c_0 = \text{IV}$

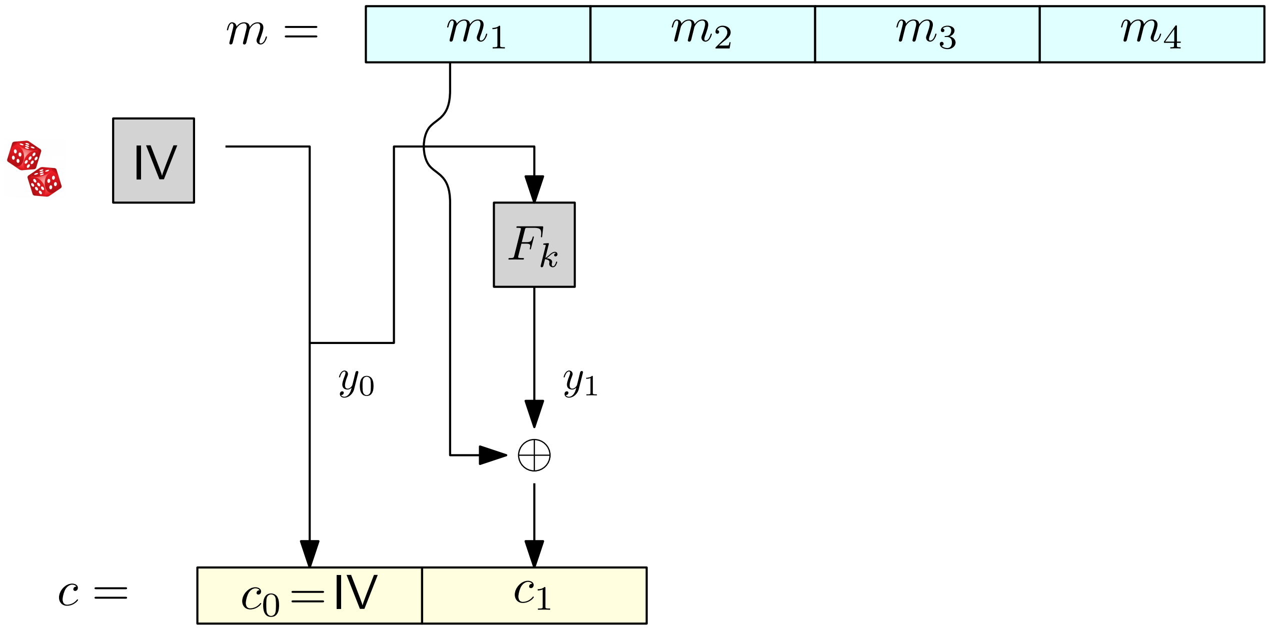
Output Feedback (OFB) mode



Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext. Let $y_0 = c_0 = \text{IV}$
- $y_i = F_k(y_{i-1})$

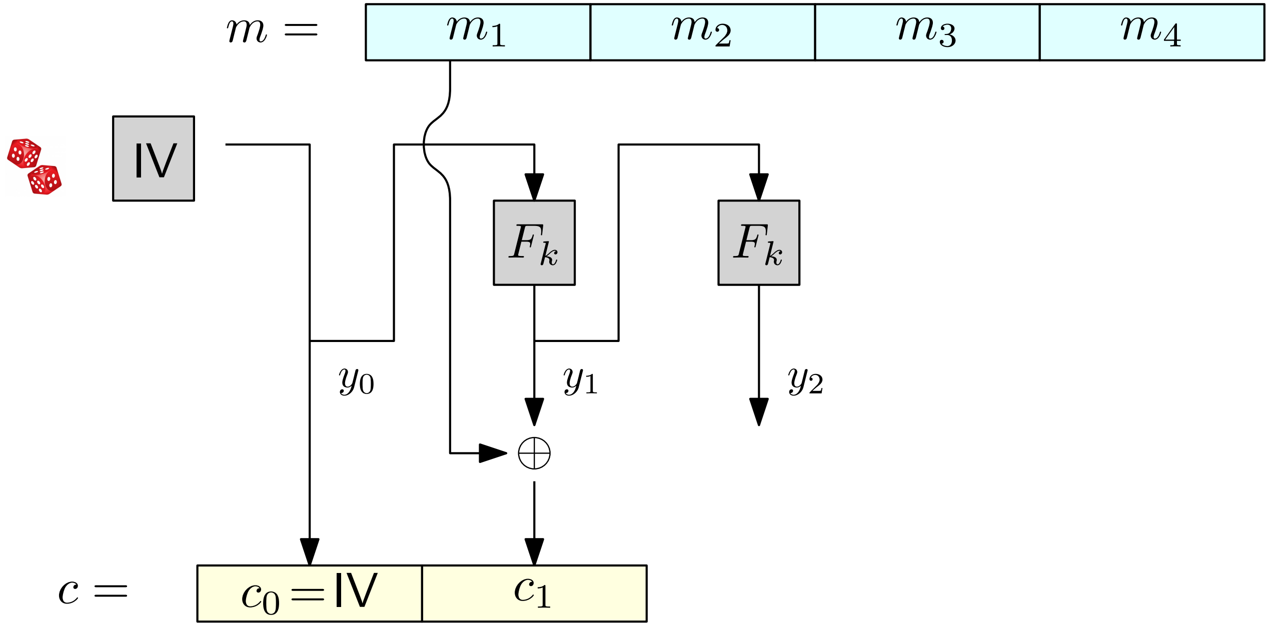
Output Feedback (OFB) mode



Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext. Let $y_0 = c_0 = \text{IV}$
- $y_i = F_k(y_{i-1})$
- $c_i = y_i \oplus m_i$

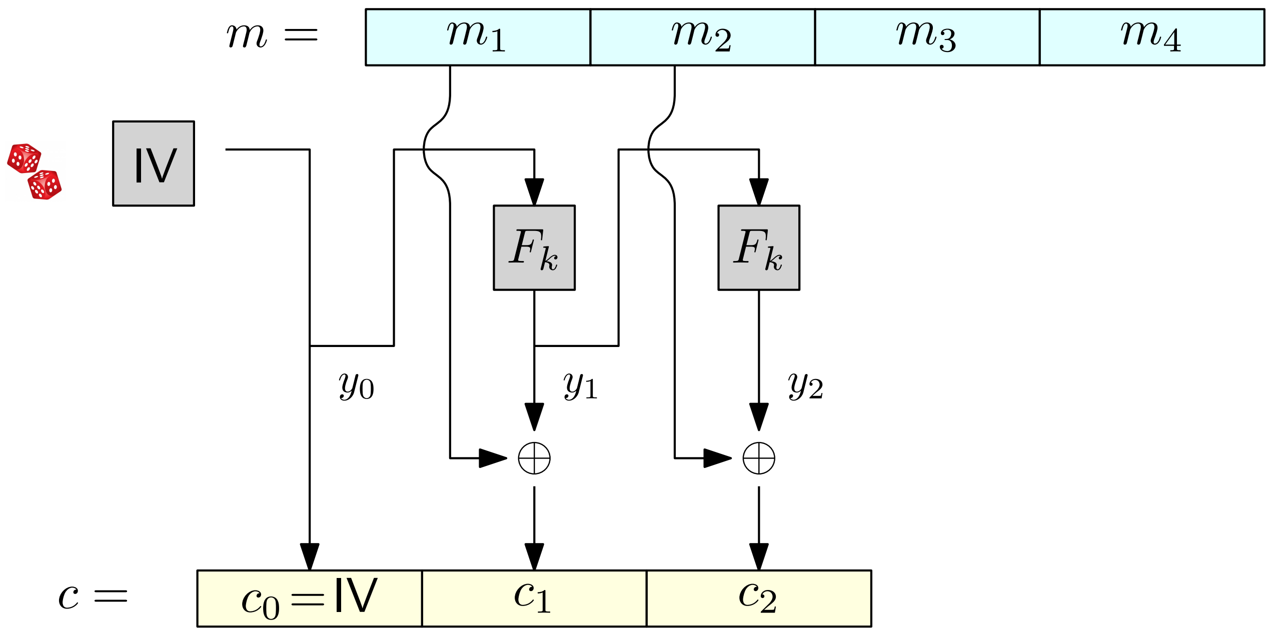
Output Feedback (OFB) mode



Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext. Let $y_0 = c_0 = \text{IV}$
- $y_i = F_k(y_{i-1})$
- $c_i = y_i \oplus m_i$

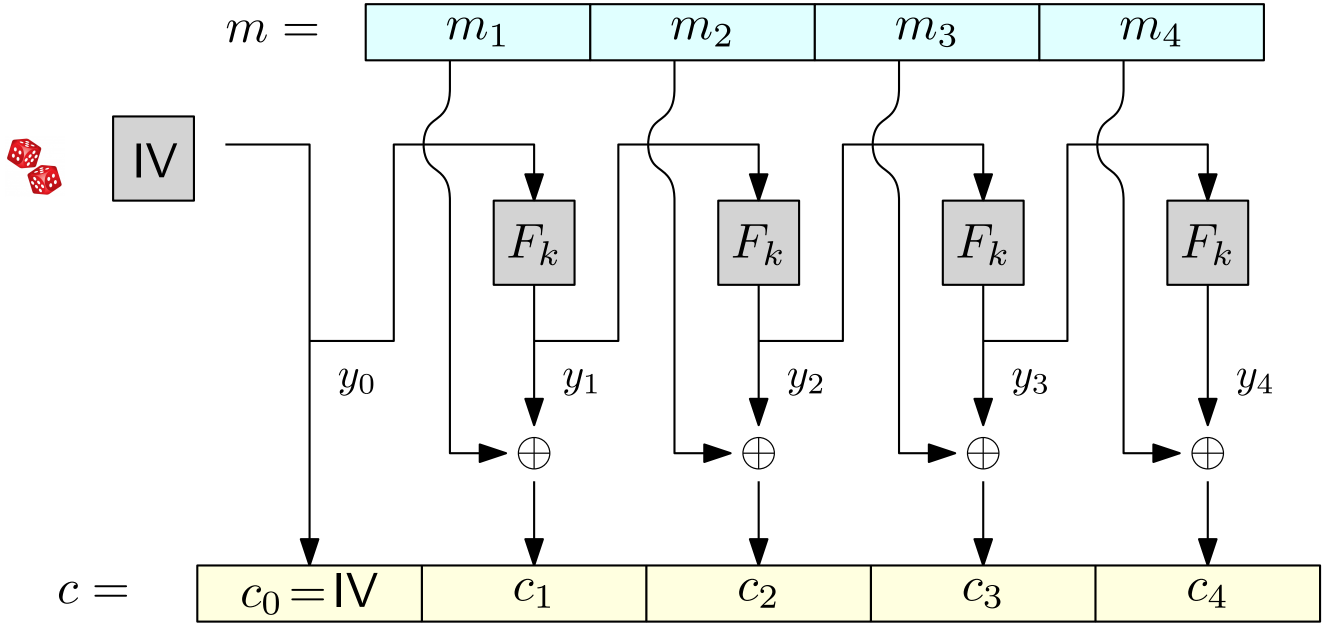
Output Feedback (OFB) mode



Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext. Let $y_0 = c_0 =$ IV
- $y_i = F_k(y_{i-1})$
- $c_i = y_i \oplus m_i$

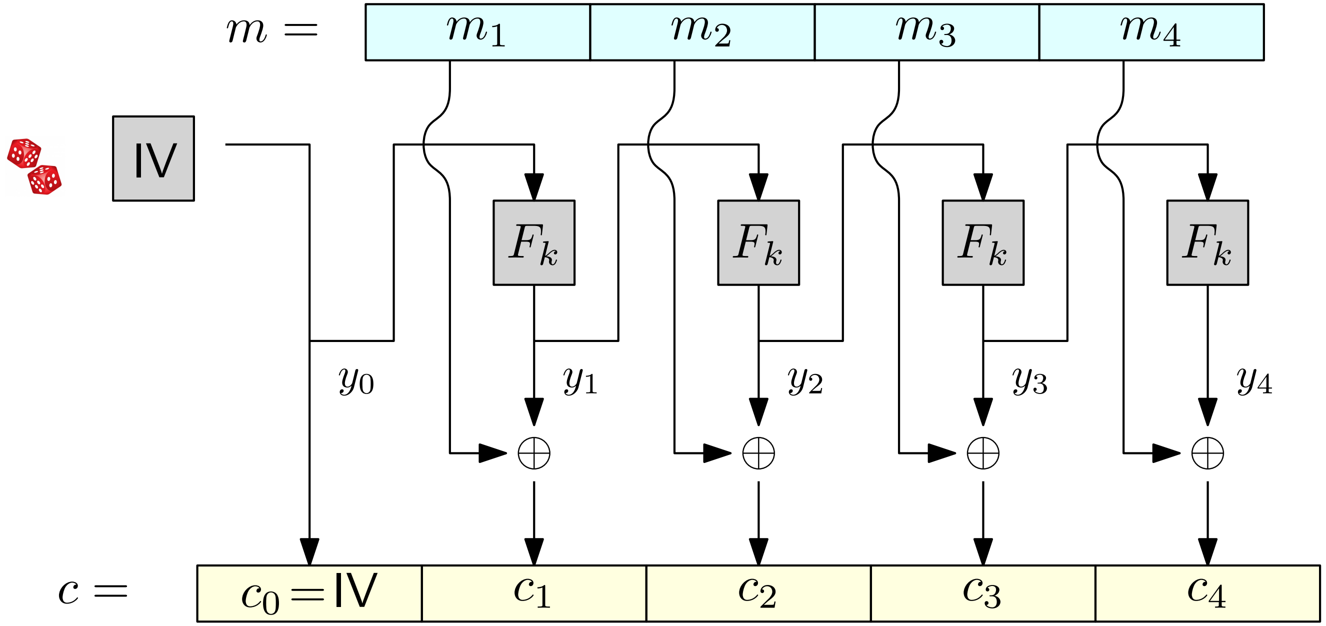
Output Feedback (OFB) mode



Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext. Let $y_0 = c_0 =$ IV
- $y_i = F_k(y_{i-1})$
- $c_i = y_i \oplus m_i$

Output Feedback (OFB) mode

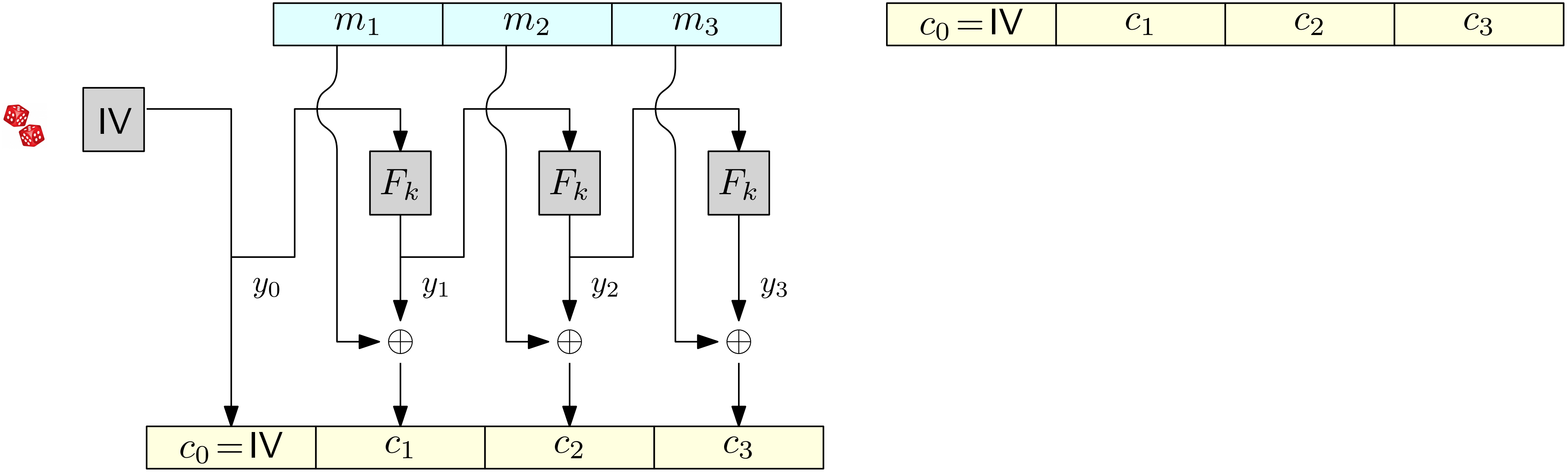


Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext. Let $y_0 = c_0 =$ IV
- $y_i = F_k(y_{i-1})$
- $c_i = y_i \oplus m_i$

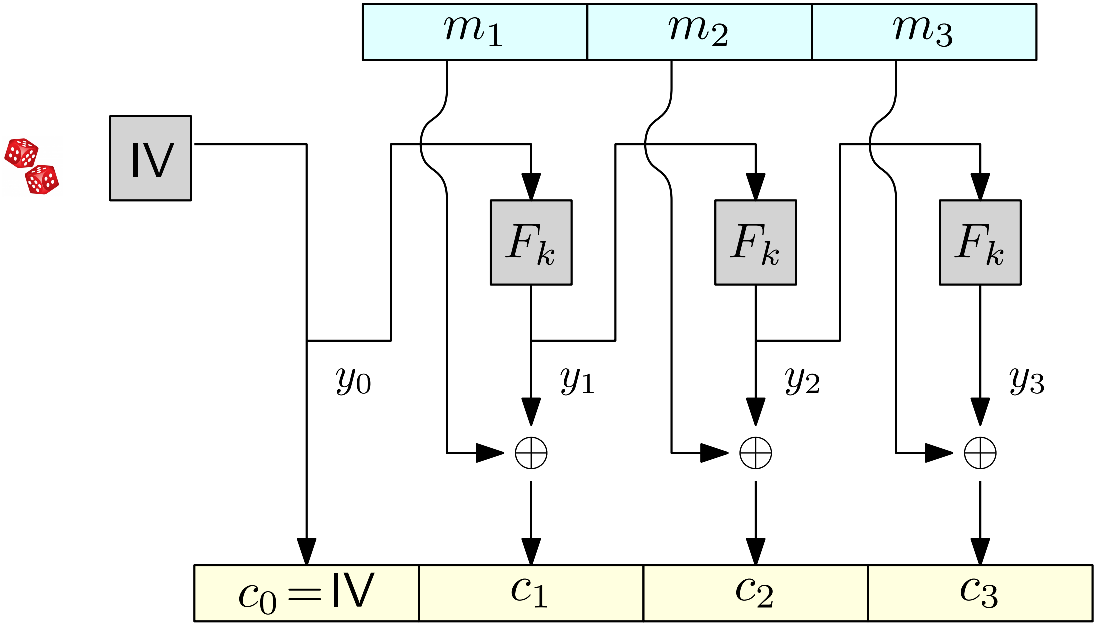
Can be thought of as a stream cipher (generate $y_1, y_2, \ldots$ and XOR it with the message)

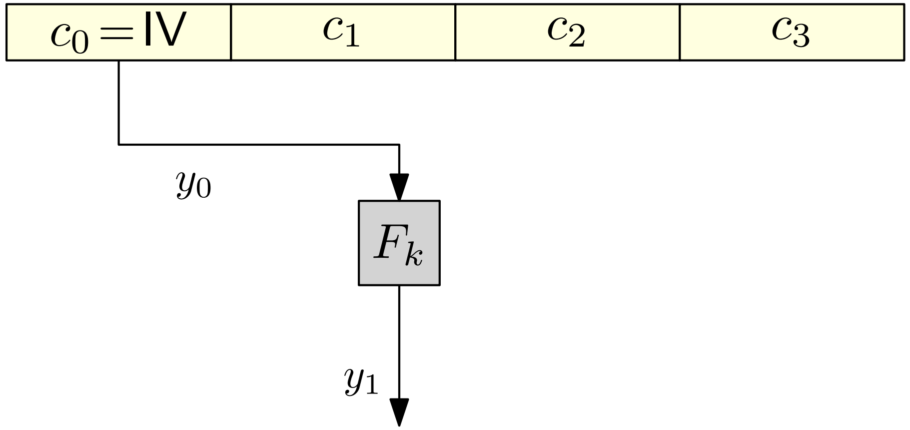
Output Feedback (OFB) mode



Decrypting:

Output Feedback (OFB) mode

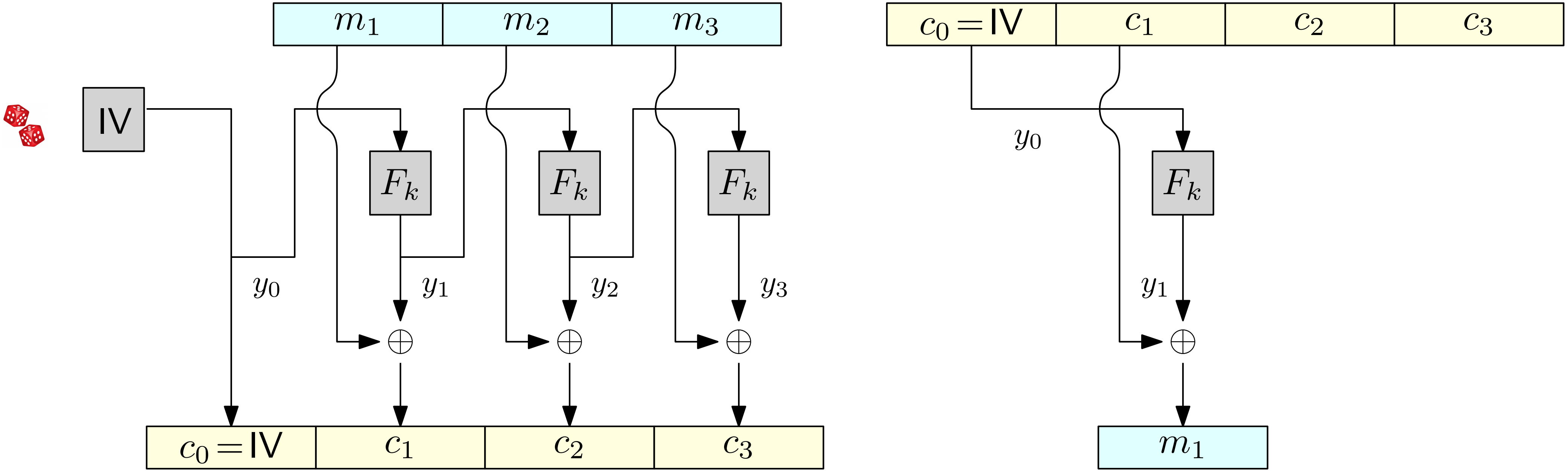




Decrypting:

- $y_0 = c_0$
- $y_i = F_k(y_{i-1})$

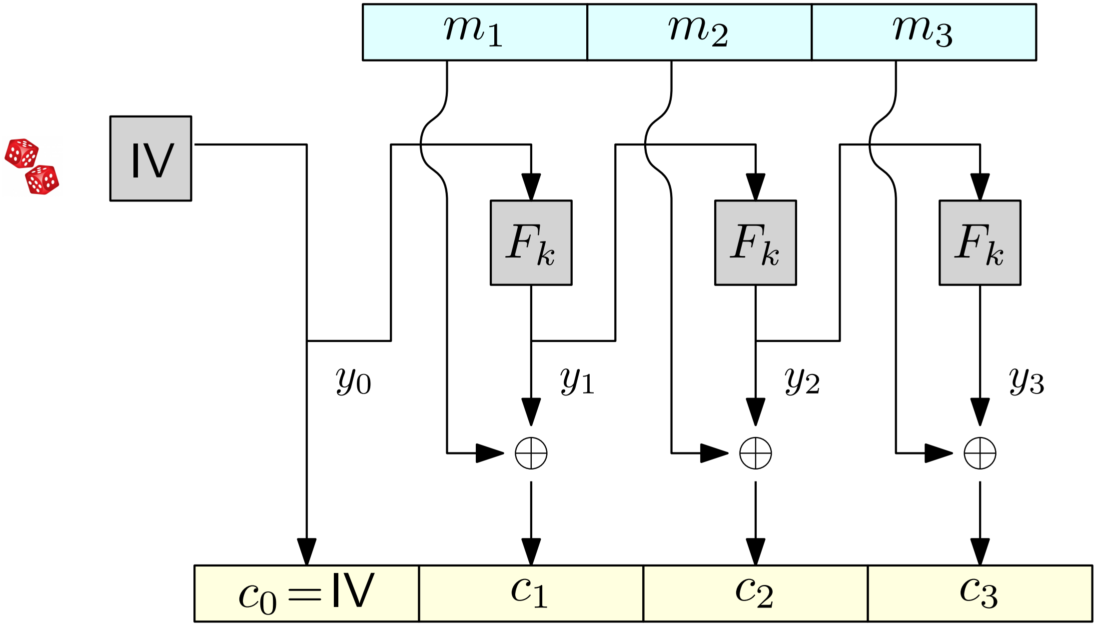
Output Feedback (OFB) mode

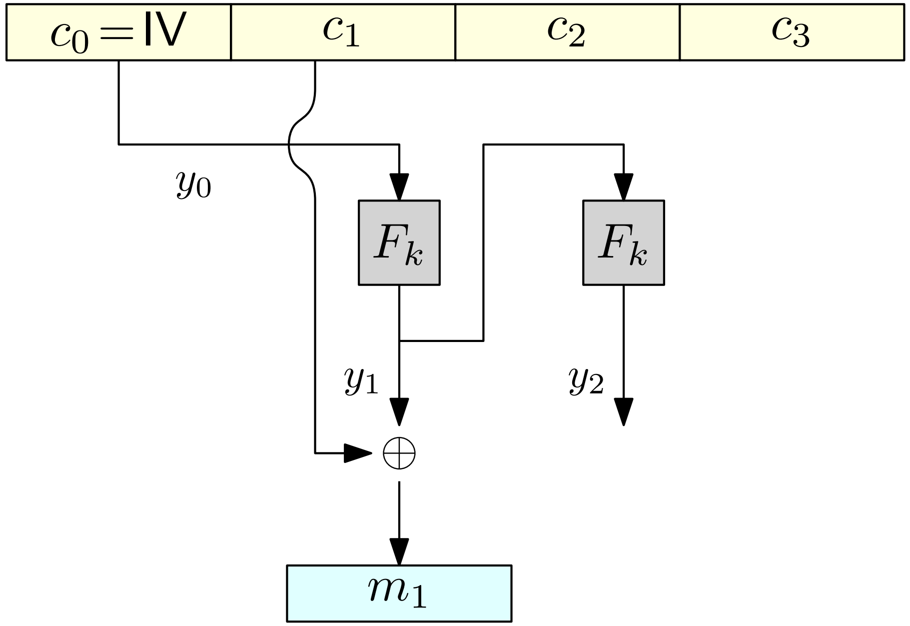


Decrypting:

- $y_0 = c_0$
- $y_i = F_k(y_{i-1})$
- $m_i = y_i \oplus c_i$

Output Feedback (OFB) mode

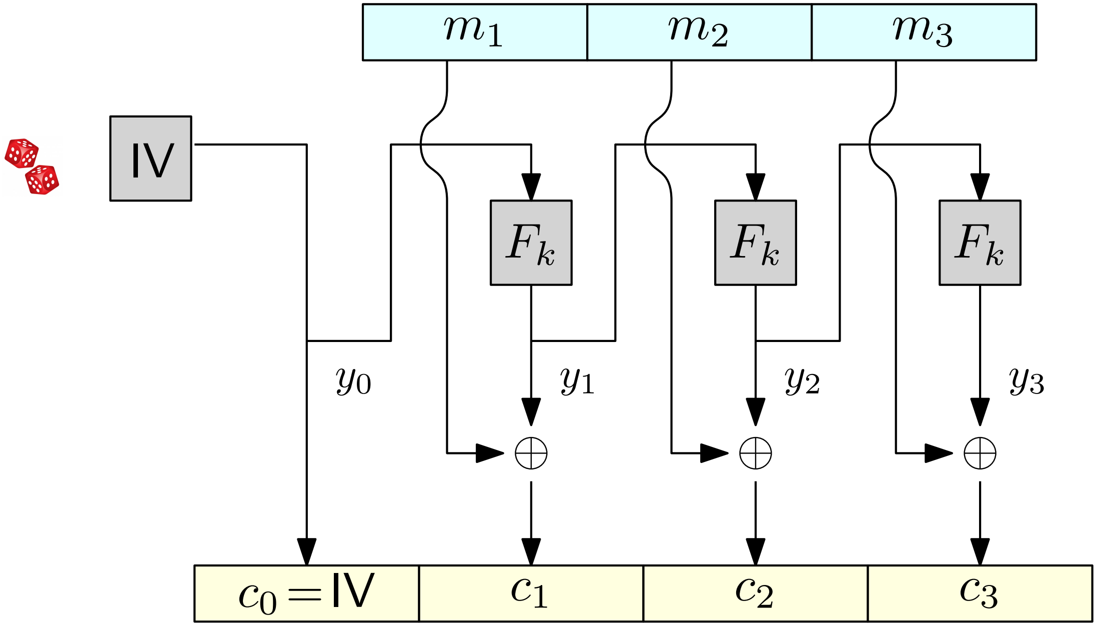


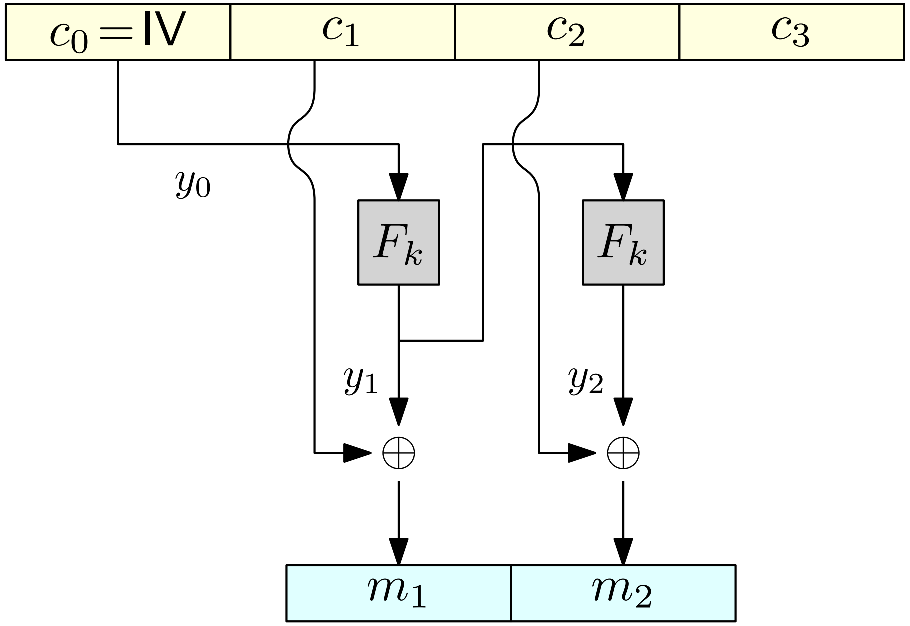


Decrypting:

- $y_0 = c_0$
- $y_i = F_k(y_{i-1})$
- $m_i = y_i \oplus c_i$

Output Feedback (OFB) mode

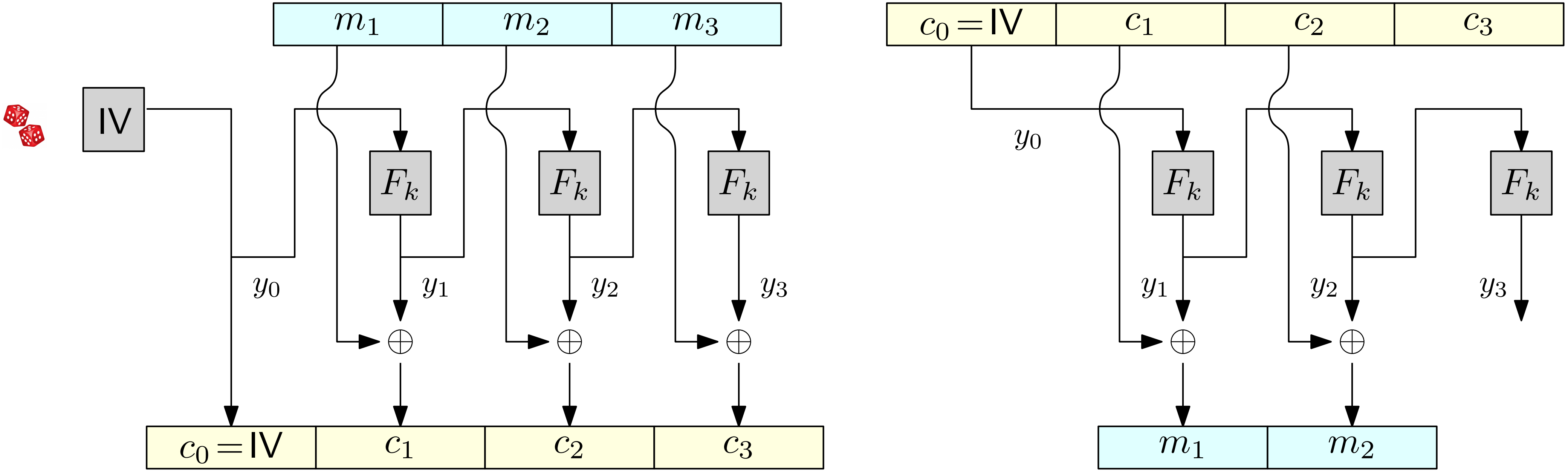




Decrypting:

- $y_0 = c_0$
- $y_i = F_k(y_{i-1})$
- $m_i = y_i \oplus c_i$

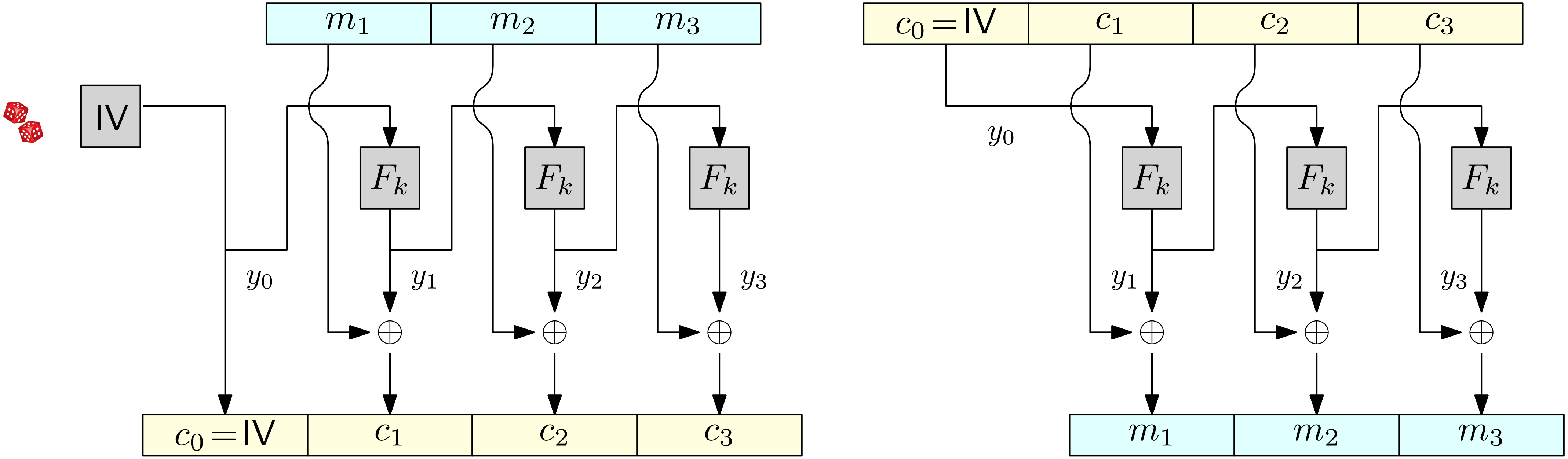
Output Feedback (OFB) mode



Decrypting:

- $y_0 = c_0$
- $y_i = F_k(y_{i-1})$
- $m_i = y_i \oplus c_i$

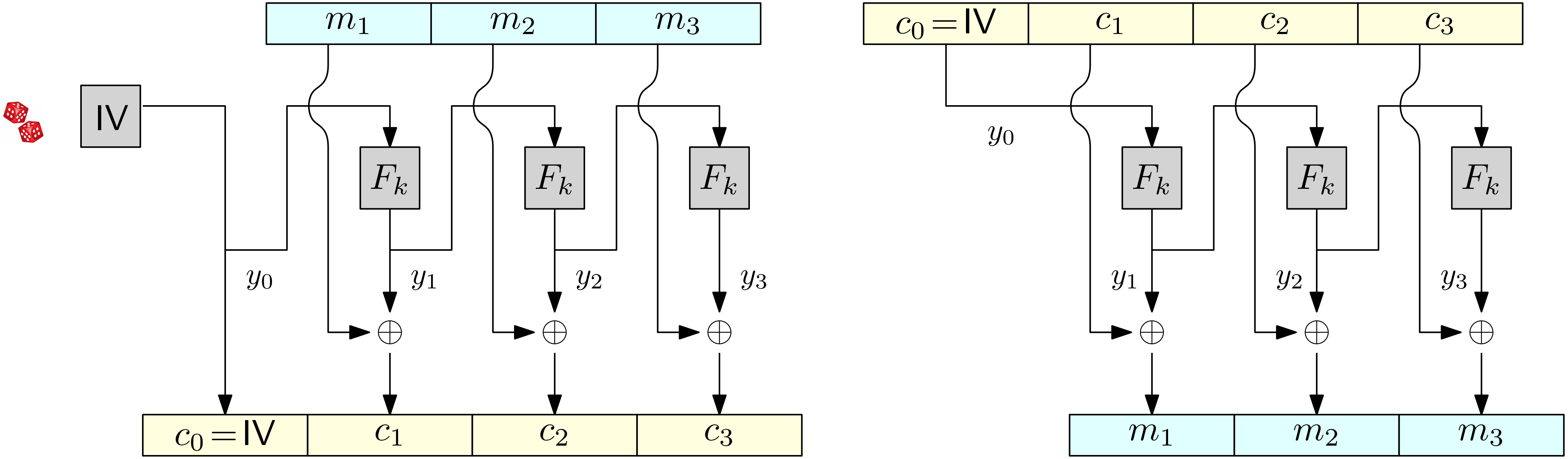
Output Feedback (OFB) mode



Decrypting:

- $y_0 = c_0$
- $y_i = F_k(y_{i-1})$
- $m_i = y_i \oplus c_i$

Output Feedback (OFB) mode



Decrypting:

- $y_0 = c_0$
- $y_i = F_k(y_{i-1})$
- $m_i = y_i \oplus c_i$

Encryption and decryption must be done sequentially

Output Feedback (OFB) mode

Encryption and decryption must be done sequentially

- An optimization: the stream $y_1, y_2, y_3, \ldots$ only depends on the IV (and the key): it can be pre-computed before the message needs to be encrypted

Output Feedback (OFB) mode

Encryption and decryption must be done sequentially

- An optimization: the stream $y_1, y_2, y_3, \ldots$ only depends on the IV (and the key): it can be pre-computed before the message needs to be encrypted

- If the last block is not full, the ciphertext can be truncated to the plaintext length

Output Feedback (OFB) mode

Encryption and decryption must be done sequentially

- An optimization: the stream $y_1, y_2, y_3, \ldots$ only depends on the IV (and the key): it can be pre-computed before the message needs to be encrypted

- If the last block is not full, the ciphertext can be truncated to the plaintext length

- F can be any PRF (not necessarily a PRP). (notice that we never used F^{-1})

Output Feedback (OFB) mode

Encryption and decryption must be done sequentially

- An optimization: the stream $y_1, y_2, y_3, \ldots$ only depends on the IV (and the key): it can be pre-computed before the message needs to be encrypted

- If the last block is not full, the ciphertext can be truncated to the plaintext length

- F can be any PRF (not necessarily a PRP). (notice that we never used F^{-1})

Is OFB mode CPA-secure?

Output Feedback (OFB) mode

Encryption and decryption must be done sequentially

- An optimization: the stream $y_1, y_2, y_3, \dots$ only depends on the IV (and the key): it can be pre-computed before the message needs to be encrypted

- If the last block is not full, the ciphertext can be truncated to the plaintext length

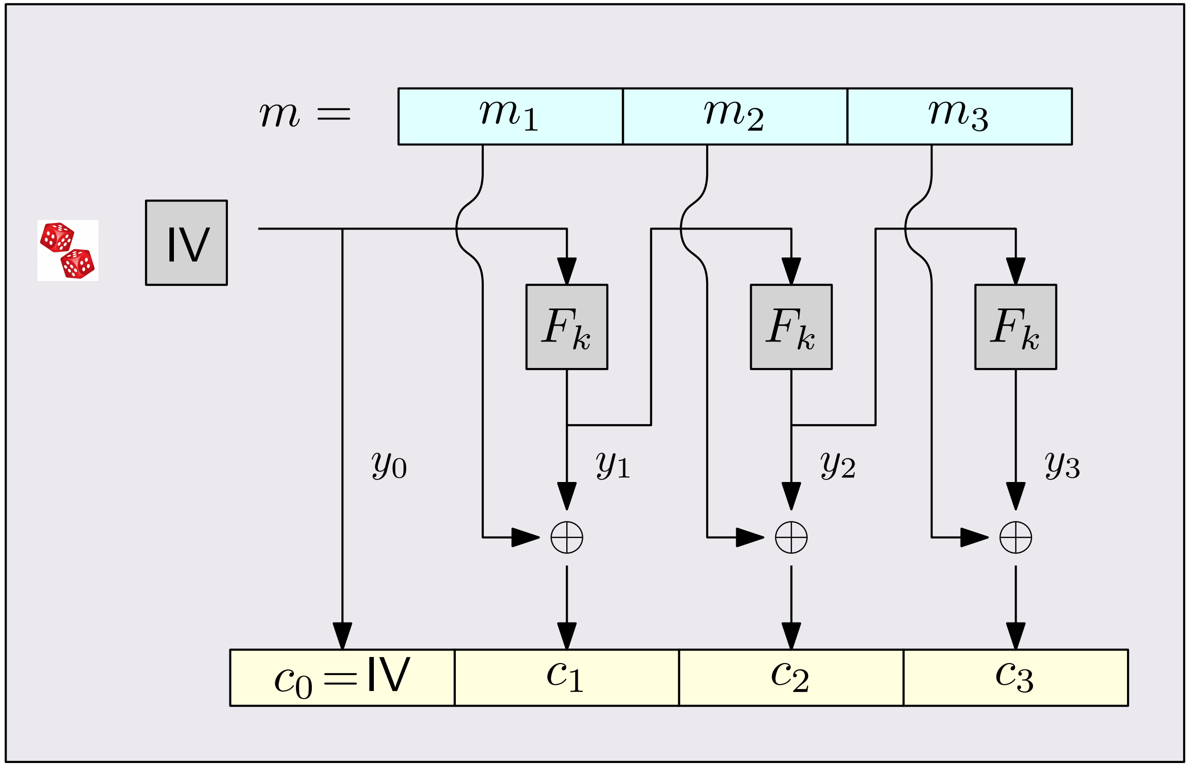
- F can be any PRF (not necessarily a PRP). (notice that we never used F^{-1})

Is OFB mode CPA-secure?

Theorem: If F is a pseudorandom function, then OFB mode is CPA-secure.

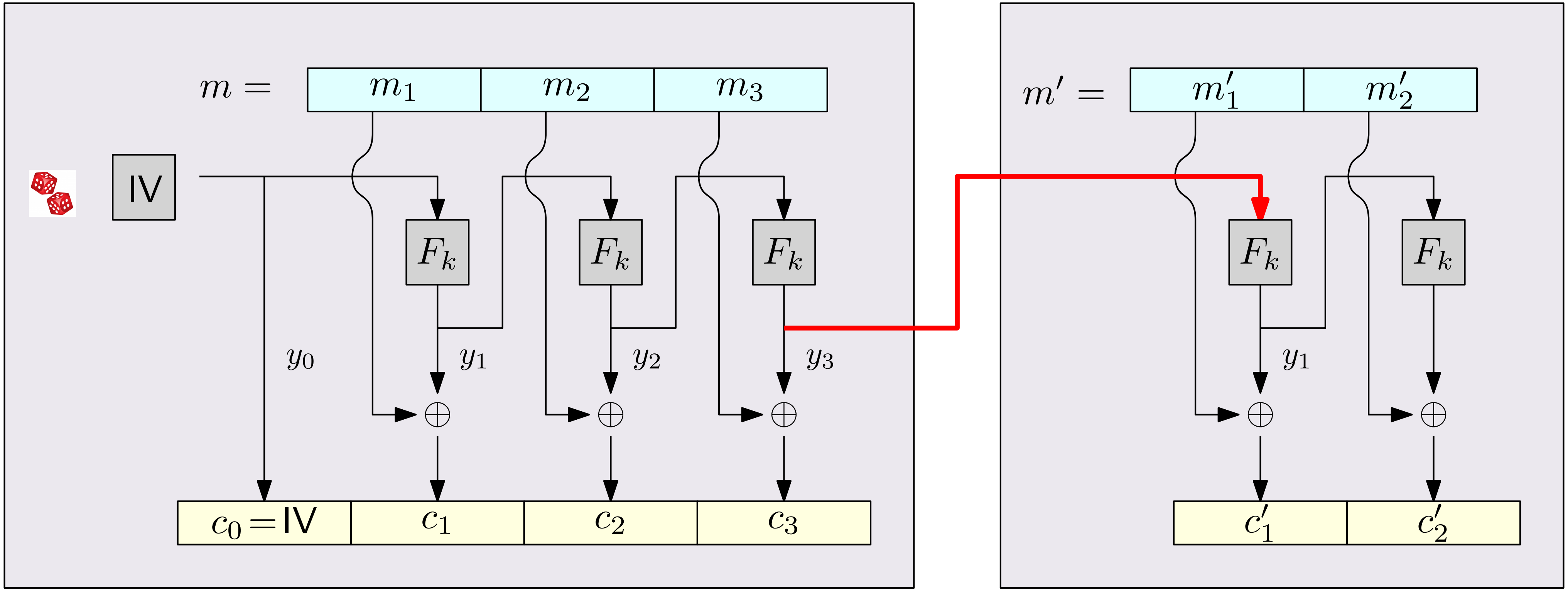
Output Feedback (OFB) mode, stateful variant

The stateful variant of OFB (the final value y_i is used in place of y_0 when the next message needs to be encrypted) is also **CPA-secure**



Output Feedback (OFB) mode, stateful variant

The stateful variant of OFB (the final value y_i is used in place of y_0 when the next message needs to be encrypted) is also **CPA-secure**



Counter (CTR) mode

Can be viewed as a stream cipher

$m =$

m_1	m_2	m_3	m_4

- Split the input to F into an IV and a counter

Counter (CTR) mode

Can be viewed as a stream cipher

$$m = \boxed{m_1}\boxed{m_2}\boxed{m_3}\boxed{m_4}$$

- Split the input to F into an IV and a counter

  For example:

  - IV $\in \{0,1\}^{3n/4}$

  - counter $\in \{0,1\}^{n/4}$

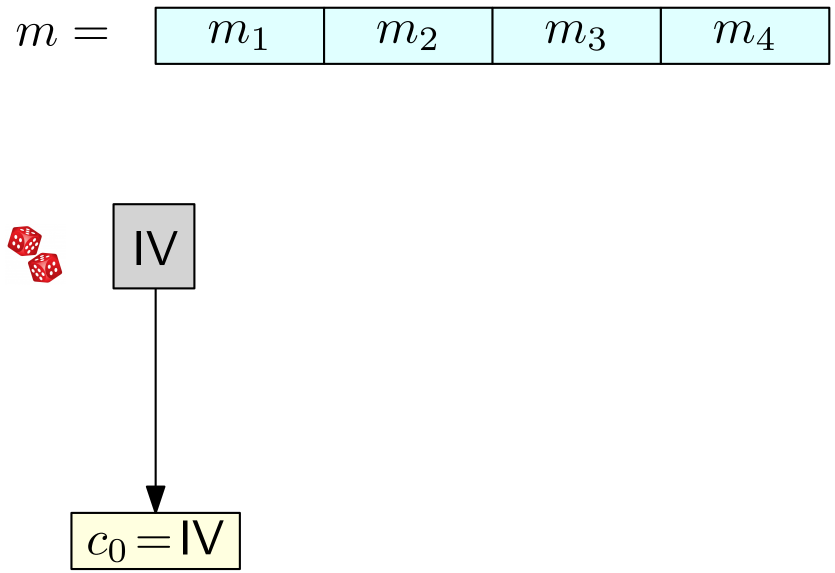
Counter (CTR) mode

Can be viewed as a stream cipher

- Split the input to F into an IV and a counter

  For example:

  - IV $\in \{0,1\}^{3n/4}$
  - counter $\in \{0,1\}^{n/4}$



Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext.

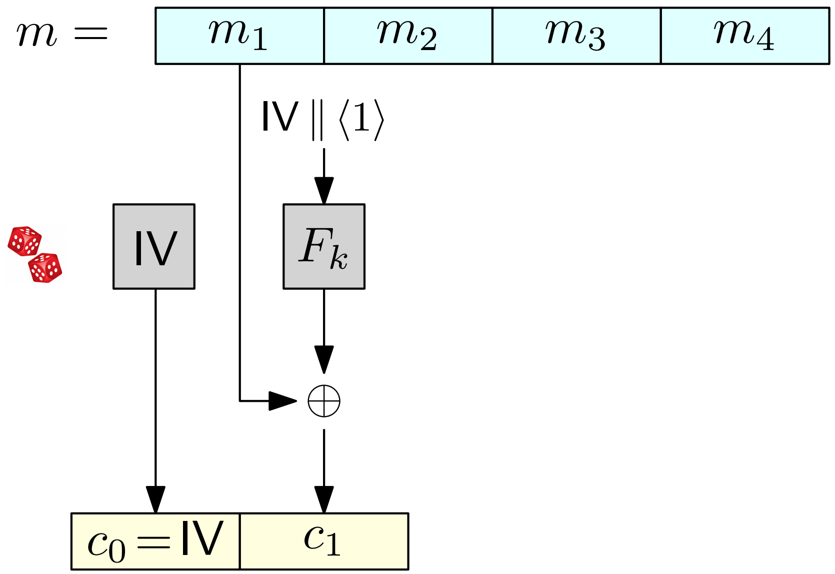
Counter (CTR) mode

Can be viewed as a stream cipher

- Split the input to F into an IV and a counter

  For example:

  - IV $\in \{0,1\}^{3n/4}$
  - counter $\in \{0,1\}^{n/4}$



Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext.
- $c_i = F_k(\mathsf{IV} \,\|\, \langle i \rangle) \oplus m_i$

Counter (CTR) mode

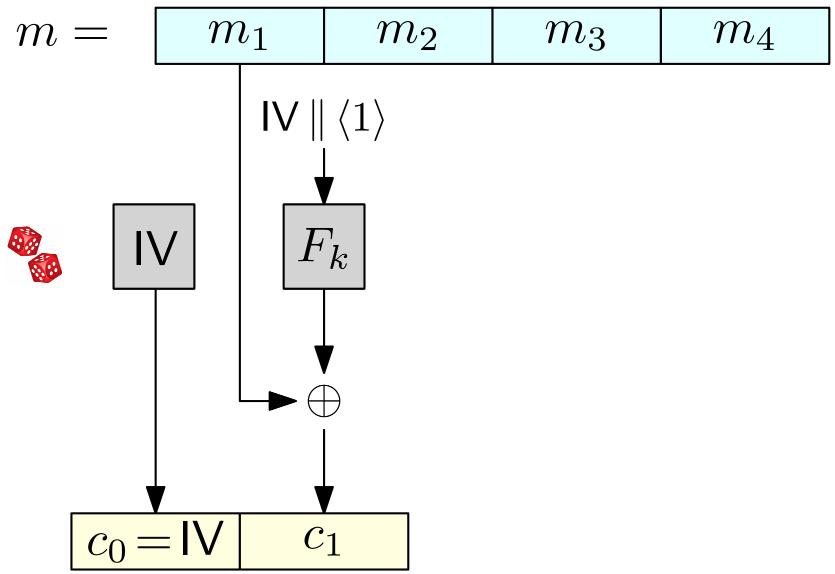
Can be viewed as a stream cipher

- Split the input to F into an IV and a counter

  For example:

  - IV $\in \{0,1\}^{3n/4}$
  - counter $\in \{0,1\}^{n/4}$

$\langle i \rangle$ Binary encoding of i



Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext.
- $c_i = F_k(\text{IV} \,\|\, \langle i \rangle) \oplus m_i$

Counter (CTR) mode

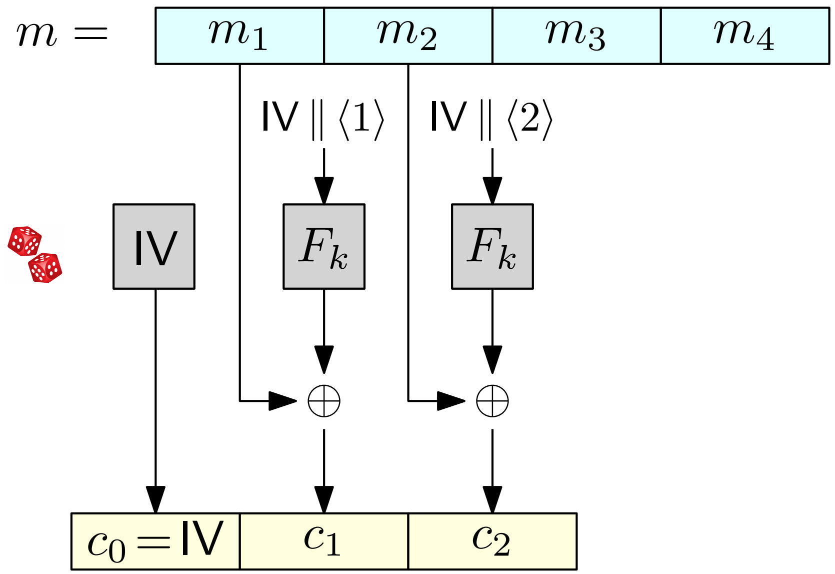
Can be viewed as a stream cipher

- Split the input to F into an IV and a counter

  For example:

  - IV $\in \{0,1\}^{3n/4}$
  - counter $\in \{0,1\}^{n/4}$

$\langle i \rangle$ Binary encoding of i



Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext.
- $c_i = F_k(\mathsf{IV} \,\|\, \langle i \rangle) \oplus m_i$

Counter (CTR) mode

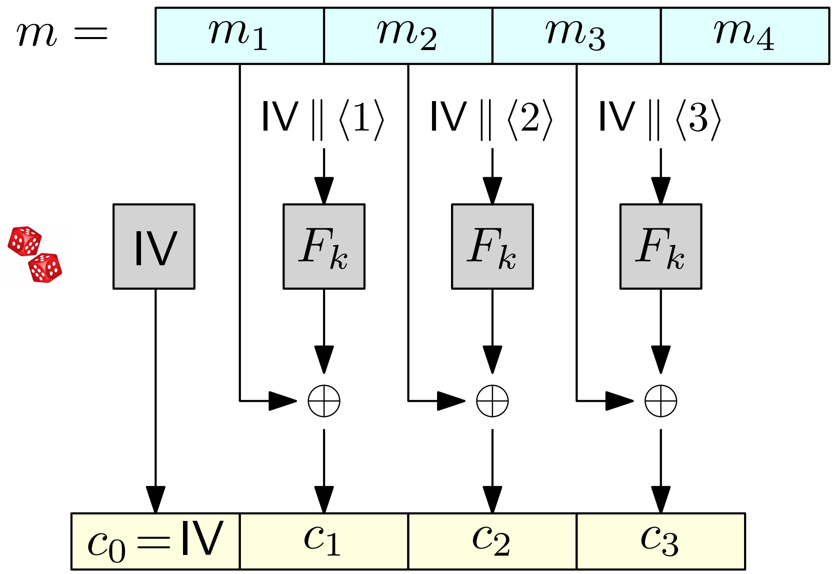
Can be viewed as a stream cipher

- Split the input to F into an IV and a counter

  For example:

  - IV $\in \{0,1\}^{3n/4}$
  - counter $\in \{0,1\}^{n/4}$

$\langle i \rangle$ Binary encoding of i



Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext.
- $c_i = F_k(\text{IV} \,\|\, \langle i \rangle) \oplus m_i$

Counter (CTR) mode

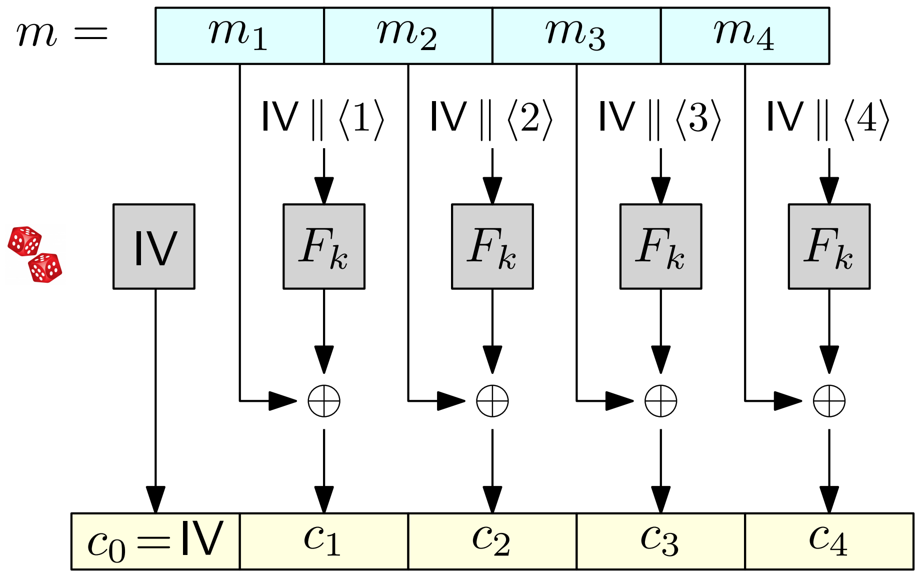
Can be viewed as a stream cipher

- Split the input to F into an IV and a counter

  For example:

  - $\mathsf{IV} \in \{0,1\}^{3n/4}$
  - counter $\in \{0,1\}^{n/4}$

$\langle i \rangle$ Binary encoding of i



Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext.
- $c_i = F_k(\mathsf{IV} \parallel \langle i \rangle) \oplus m_i$

Counter (CTR) mode

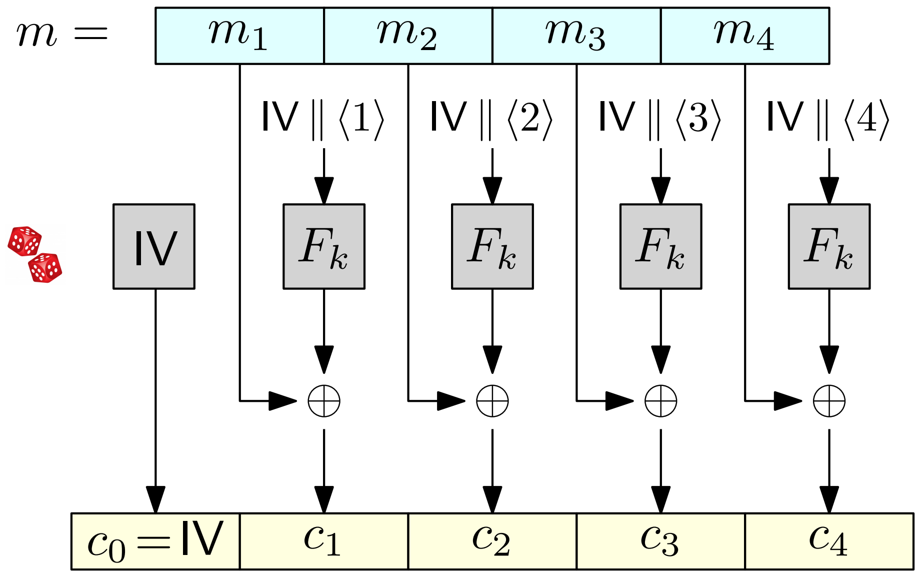
Can be viewed as a stream cipher

- Split the input to F into an IV and a counter

  For example:

  - IV $\in \{0,1\}^{3n/4}$
  - counter $\in \{0,1\}^{n/4}$

$\langle i \rangle$ Binary encoding of i



Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext.
- $c_i = F_k(\text{IV} \| \langle i \rangle) \oplus m_i$

Decrypting:

- Set the IV to the first block c_0 of the ciphertext.

Counter (CTR) mode

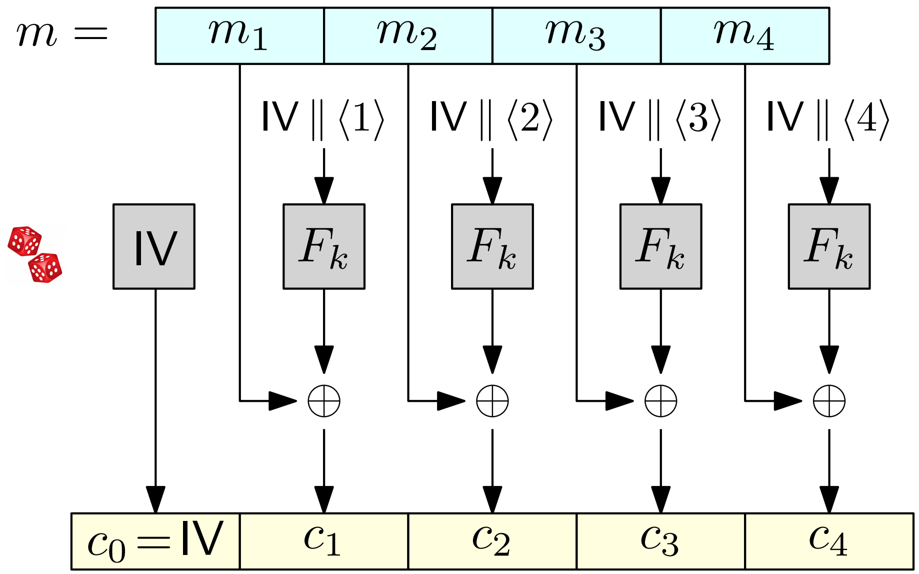
Can be viewed as a stream cipher

- Split the input to F into an IV and a counter

  For example:

  - $\mathsf{IV} \in \{0,1\}^{3n/4}$
  - counter $\in \{0,1\}^{n/4}$

$\langle i \rangle$ Binary encoding of i



Encrypting:

- A random IV is chosen and sent as the first block c_0 of the ciphertext.
- $c_i = F_k(\mathsf{IV} \parallel \langle i \rangle) \oplus m_i$

Decrypting:

- Set the IV to the first block c_0 of the ciphertext.
- $m_i = F_k(\mathsf{IV} \parallel \langle i \rangle) \oplus c_i$

Counter (CTR) mode

- The length of the IV affects the security
- The length of the counter controls how many blocks can be sent with the same IV

Counter (CTR) mode

- The length of the IV affects the security
- The length of the counter controls how many blocks can be sent with the same IV
- Both encryption and decryption can be done in parallel!

Counter (CTR) mode

- The length of the IV affects the security
- The length of the counter controls how many blocks can be sent with the same IV
- Both encryption and decryption can be done in parallel!
- If the last block is not full, the ciphertext can be truncated to the plaintext length (no padding needed)

Counter (CTR) mode

- The length of the IV affects the security
- The length of the counter controls how many blocks can be sent with the same IV
- Both encryption and decryption can be done in parallel!
- If the last block is not full, the ciphertext can be truncated to the plaintext length (no padding needed)
- F can be any PRF (not necessarily a PRP) (notice that we never used F^{-1})

Counter (CTR) mode

- The length of the IV affects the security
- The length of the counter controls how many blocks can be sent with the same IV
- Both encryption and decryption can be done in parallel!
- If the last block is not full, the ciphertext can be truncated to the plaintext length (no padding needed)
- F can be any PRF (not necessarily a PRP) (notice that we never used F^{-1})

Is CTR mode CPA-secure?

Counter (CTR) mode

- The length of the IV affects the security
- The length of the counter controls how many blocks can be sent with the same IV
- Both encryption and decryption can be done in parallel!
- If the last block is not full, the ciphertext can be truncated to the plaintext length (no padding needed)
- F can be any PRF (not necessarily a PRP) (notice that we never used F^{-1})

Is CTR mode CPA-secure?

Theorem: If F is a pseudorandom function, then CTR mode is CPA-secure.

Counter (CTR) mode

- The length of the IV affects the security
- The length of the counter controls how many blocks can be sent with the same IV
- Both encryption and decryption can be done in parallel!
- If the last block is not full, the ciphertext can be truncated to the plaintext length (no padding needed)
- F can be any PRF (not necessarily a PRP) (notice that we never used F^{-1})

Is CTR mode CPA-secure?

Theorem: If F is a pseudorandom function, then CTR mode is CPA-secure.

- Remains secure even if IVs are not chosen u.a.r., in fact it suffices that IVs never repeat

$$\text{IV} = 00\ldots000,\ 00\ldots001,\ 00\ldots010,\ 00\ldots011,\ \ldots$$