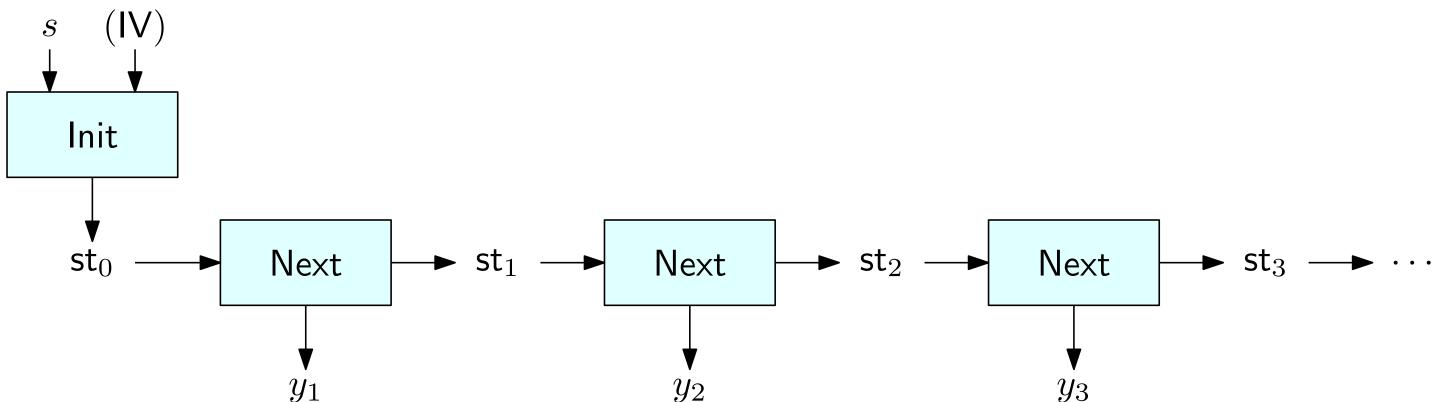
Stream ciphers (reminder)

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)

• Stands for Rivest Cipher 4

• Designed for performance in software

RC4



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



- 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



- 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



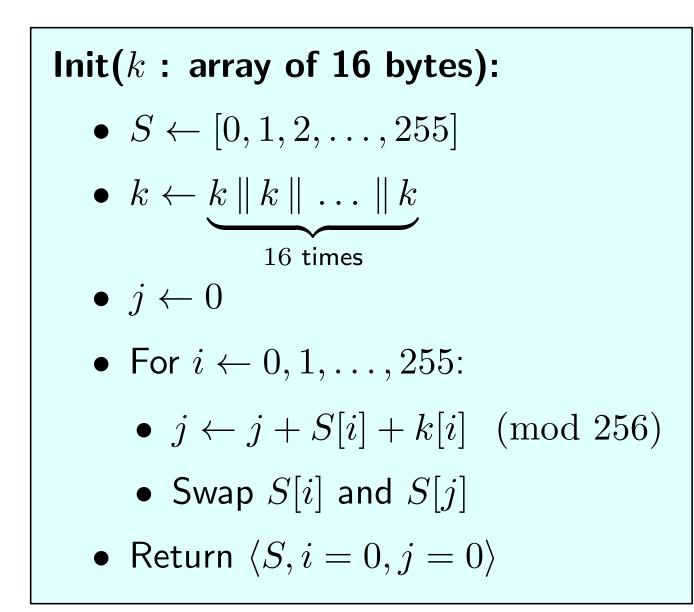
The state consists of:

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

R(24

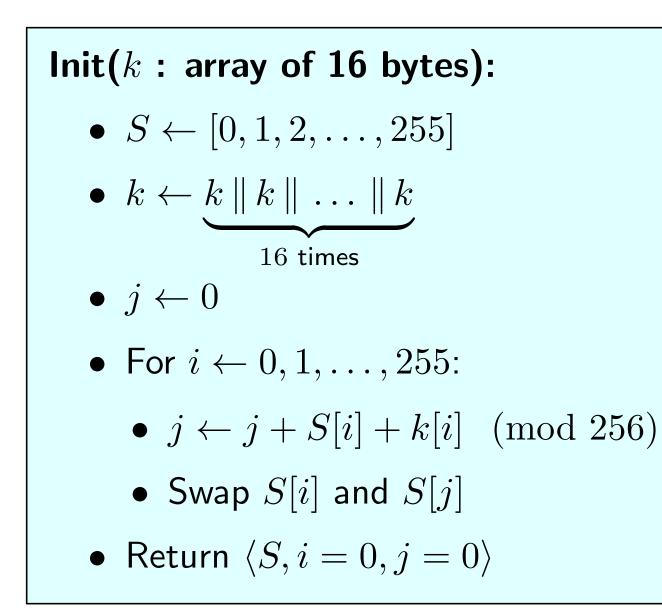
The state consists of:

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



The state consists of:

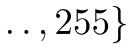
- An array S of 256 bytes, which will always be a permutation of $\{0, \ldots, 255\}$
- A pair of integers $i, j \in \{0, \dots, 255\}$

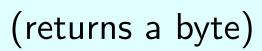


Next(st = $\langle S, i, j \rangle$):

- $i \leftarrow i+1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap S[i] and S[j]
- $t = S[i] + S[j] \pmod{256}$

•
$$y \leftarrow S[t]$$









Test vectors

Key length: 128 bits.

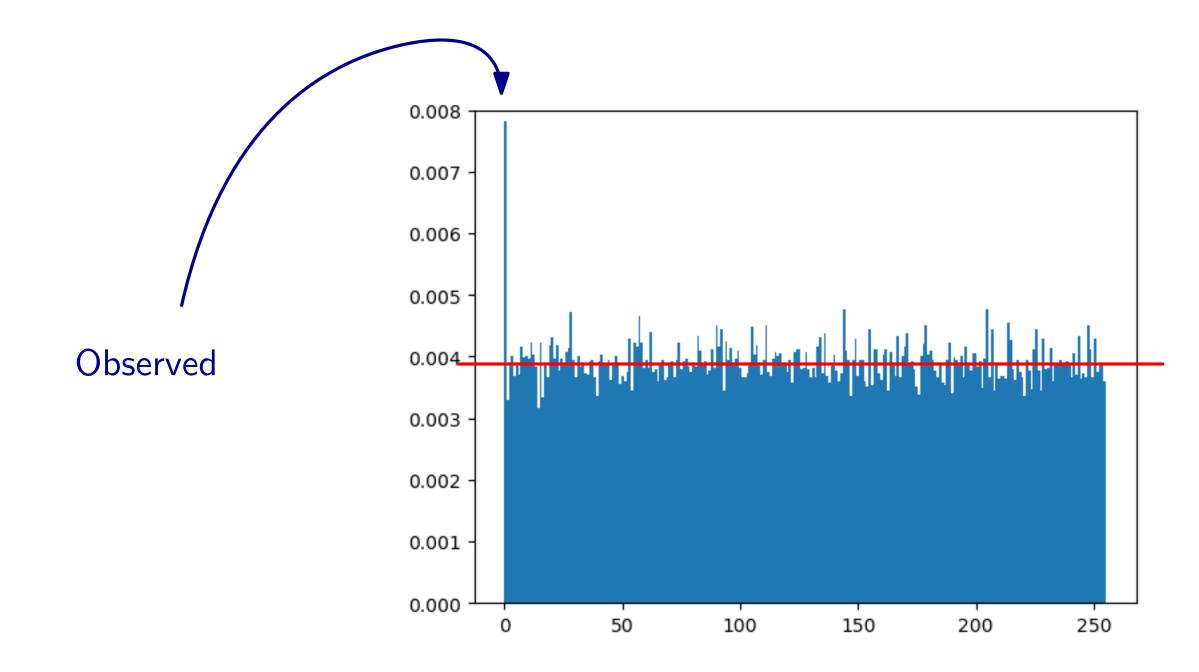
key: 0x0102030405060708090a0b0c0d0e0f10

DEC	0	HEX	0:	9a	c7	сс	9a	60	9d	1e	f7	b2	93	28	99
DEC	16	HEX	10:	52	48	c4	95	90	14	12	6a	6e	8a	84	f1
DEC	240	HEX	f0:	06	59	02	e4	b6	20	f6	сс	36	c8	58	9f
DEC	256	HEX	100:	d3	9d	56	6b	c6	bc	e3	01	07	68	15	15
DEC	496	HEX	1f0:	b6	d1	e6	c4	a5	e4	77	1c	ad	79	53	8d
DEC	512	HEX	200:	c6	8c	1d	5c	55	9a	97	41	23	df	1d	bc
DEC	752	HEX	2f0:	c5	ec	f8	8d	e8	97	fd	57	fe	d3	01	70
DEC	768	HEX	300:	ec	cb	e1	3d	e1	fc	c9	1c	11	a0	b2	6c
DEC	1008	HEX	3f0:	e7	a7	25	74	f8	78	2a	e2	6a	ab	cf	9e
DEC	1024	HEX	400:	bd	f0	32	4e	60	83	dc	c6	d3	ce	dd	3с
DEC	1520	HEX	5f0:	b4	01	10	c4	19	0b	56	22	a9	61	16	b0
DEC	1536	HEX	600:	ff	a0	b5	14	64	7e	с0	4f	63	06	b8	92
DEC	2032	HEX	7f0:	d0	3d	1b	с0	3c	d3	3d	70	df	f9	fa	5d
DEC	2048	HEX	800:	8a	44	12	64	11	ea	a7	8b	d5	1e	8d	87
DEC	3056	HEX	bf0:	fa	be	b7	60	28	ad	e2	d0	e4	87	22	e4
DEC	3072	HEX	c00:	с0	5d	88	ab	d5	03	57	f9	35	a6	3c	59
DEC	4080	HEX	ff0:	ff	38	26	5c	16	42	c1	ab	e8	d3	c2	fe
DEC	4096	HEX	1000:	a3	6a	4c	30	1a	e8	ac	13	61	0c	cb	c1
	DEC DEC DEC DEC DEC DEC DEC DEC DEC DEC	DEC16DEC240DEC256DEC496DEC512DEC752DEC768DEC1008DEC1024DEC1520DEC1536DEC2032DEC2048DEC3056DEC3072DEC4080	DEC16HEXDEC240HEXDEC256HEXDEC496HEXDEC512HEXDEC752HEXDEC768HEXDEC1008HEXDEC1024HEXDEC1520HEXDEC2032HEXDEC2048HEXDEC3056HEXDEC3072HEXDEC4080HEX	 DEC 16 HEX 10: DEC 240 HEX f0: DEC 256 HEX 100: DEC 496 HEX 1f0: DEC 512 HEX 200: DEC 752 HEX 2f0: DEC 768 HEX 300: DEC 1008 HEX 3f0: DEC 1024 HEX 400: DEC 1520 HEX 5f0: DEC 1536 HEX 600: DEC 2032 HEX 7f0: DEC 2048 HEX 800: DEC 3056 HEX bf0: DEC 3072 HEX c00: 	DEC16HEX10:52DEC240HEXf0:06DEC256HEX100:d3DEC496HEX1f0:b6DEC512HEX200:c6DEC752HEX2f0:c5DEC768HEX300:ecDEC1008HEX3f0:e7DEC1024HEX400:bdDEC1520HEX5f0:b4DEC1536HEX600:ffDEC2032HEX7f0:d0DEC3056HEXbf0:faDEC3072HEXc00:c0DEC4080HEXff0:ff	DEC16HEX10:5248DEC240HEXf0:0659DEC256HEX100:d39dDEC496HEX1f0:b6d1DEC512HEX200:c68cDEC752HEX2f0:c5ecDEC768HEX300:eccbDEC1008HEX3f0:e7a7DEC1024HEX400:bdf0DEC1520HEX5f0:b401DEC1536HEX600:ffa0DEC2032HEX7f0:d03dDEC3056HEXbf0:fabeDEC3072HEXc00:c05dDEC4080HEXff0:ff38	DEC16HEX10:5248c4DEC240HEXf0:065902DEC256HEX100:d39d56DEC496HEX1f0:b6d1e6DEC512HEX200:c68c1dDEC752HEX2f0:c5ecf8DEC768HEX300:eccbe1DEC1008HEX3f0:e7a725DEC1024HEX400:bdf032DEC1520HEX5f0:b40110DEC1536HEX600:ffa0b5DEC2032HEX7f0:d03d1bDEC3056HEXbf0:fabeb7DEC3072HEXc00:c05d88DEC4080HEXff0:ff3826	DEC16HEX10:5248c495DEC240HEXf0:065902e4DEC256HEX100:d39d566bDEC496HEX1f0:b6d1e6c4DEC512HEX200:c68c1d5cDEC752HEX2f0:c5ecf88dDEC768HEX300:eccbe13dDEC1008HEX3f0:e7a72574DEC1024HEX400:bdf0324eDEC1520HEX5f0:b40110c4DEC1536HEX800:ffa0b514DEC2032HEX7f0:d03d1bc0DEC3056HEXbf0:fabeb760DEC3072HEXc00:c05d88abDEC4080HEXff0:ff38265c	DEC16HEX10:5248c49590DEC240HEXf0:065902e4b6DEC256HEX100:d39d566bc6DEC496HEX1f0:b6d1e6c4a5DEC512HEX200:c68c1d5c55DEC752HEX2f0:c5ecf88de8DEC768HEX300:eccbe13de1DEC1008HEX3f0:e7a72574f8DEC1024HEX400:bdf0324e60DEC1536HEX5f0:b40110c419DEC2032HEX7f0:d03d1bc03cDEC2048HEX800:8a44126411DEC3056HEXbf0:fabeb76028DEC3072HEXc00:c05d88abd5DEC4080HEXff0:ff38265c16	DEC16HEX10:5248c4959014DEC240HEXf0:065902e4b620DEC256HEX100:d39d566bc6bcDEC496HEX1f0:b6d1e6c4a5e4DEC512HEX200:c68c1d5c559aDEC752HEX2f0:c5ecf88de897DEC768HEX300:eccbe13de1fcDEC1024HEX3f0:e7a72574f878DEC1024HEX5f0:b40110c4190bDEC1536HEX600:ffa0b514647eDEC2032HEX7f0:d03d1bc03cd3DEC2048HEX800:8a44126411eaDEC3056HEXbf0:fabeb76028adDEC3072HEXc00:c05d88abd503DEC4080HEXff0:ff38265c1642	DEC16HEX10:5248c495901412DEC240HEXf0:065902e4b620f6DEC256HEX100:d39d566bc6bce3DEC496HEX1f0:b6d1e6c4a5e477DEC512HEX200:c68c1d5c559a97DEC752HEX2f0:c5ecf88de897fdDEC768HEX300:eccbe13de1fcc9DEC1008HEX3f0:e7a72574f8782aDEC1024HEX400:bdf0324e6083dcDEC1520HEX5f0:b40110c4190b56DEC1536HEX600:ffa0b514647ec0DEC2032HEX7f0:d03d1bc03cd33dDEC2048HEX800:8a44126411eaa7DEC3056HEXbf0:fabeb76028ade2DEC3072HEXc00:c05d88abd50357	DEC16HEX10:5248c4959014126aDEC240HEXf0:065902e4b620f6ccDEC256HEX100:d39d566bc6bce301DEC496HEX1f0:b6d1e6c4a5e4771cDEC512HEX200:c68c1d5c559a9741DEC752HEX2f0:c5ecf88de897fd57DEC768HEX300:eccbe13de1fcc91cDEC1008HEX3f0:e7a72574f8782ae2DEC1024HEX400:bdf0324e6083dcc6DEC1520HEX5f0:b40110c4190b5622DEC1536HEX600:ffa0b514647ec04fDEC2032HEX7f0:d03d1bc03cd33d70DEC3056HEXbf0:fabeb76028ade2d0DEC3072HEXc00:c05d88abd50357f9	DEC16HEX10:5248c4959014126a6eDEC240HEXf0:065902e4b620f6cc36DEC256HEX100:d39d566bc6bce30107DEC496HEX1f0:b6d1e6c4a5e4771cadDEC512HEX200:c68c1d5c559a974123DEC752HEX2f0:c5ecf88de897fd57feDEC768HEX300:eccbe13de1fcc91c11DEC1008HEX3f0:e7a72574f8782ae26aDEC1024HEX400:bdf0324e6083dcc6d3DEC1520HEX5f0:b40110c4190b5622a9DEC1536HEX800:ff <a0< td="">b514647ec04f63DEC2032HEX7f0:d03d1bc03cd33d70dfDEC3056HEXbf0:fabeb76028ade2d0e4<t< td=""><td>DEC16HEX10:5248c4959014126a6e8aDEC240HEXf0:065902e4b620f6cc36c8DEC256HEX100:d39d566bc6bce3010768DEC496HEX1f0:b6d1e6c4a5e4771cad79DEC512HEX200:c68c1d5c559a974123dfDEC752HEX2f0:c5ecf88de897fd57fed3DEC768HEX300:eccbe13de1fcc91c11a0DEC1024HEX3f0:e7a72574f8782ae26aabDEC1520HEX5f0:b40110c4190b5622a961DEC1536HEX600:ff<a0< td="">3d1bc03cd33d70dff9DEC2032HEX7f0:d03d1bc03cd33d70dff9DEC2048HEX800:8a44126411eaa78bd51eDEC<</a0<></td><td>DEC 16 HEX 10: 52 48 c4 95 90 14 12 6a 6e 8a 84 DEC 240 HEX f0: 06 59 02 e4 b6 20 f6 cc 36 c8 58 DEC 256 HEX 100: d3 9d 56 6b c6 bc e3 01 07 68 15 DEC 496 HEX 1f0: b6 d1 e6 c4 a5 e4 77 1c ad 79 53 DEC 512 HEX 200: c6 8c 1d 5c 55 9a 97 41 23 df 1d DEC 752 HEX 2f0: c5 ec f8 8d e8 97 fd 57 fe d3 01 DEC 768 HEX 300: ec cb e1 3d e1 fc c9 1c 11 a0 <</td></t<></a0<>	DEC16HEX10:5248c4959014126a6e8aDEC240HEXf0:065902e4b620f6cc36c8DEC256HEX100:d39d566bc6bce3010768DEC496HEX1f0:b6d1e6c4a5e4771cad79DEC512HEX200:c68c1d5c559a974123dfDEC752HEX2f0:c5ecf88de897fd57fed3DEC768HEX300:eccbe13de1fcc91c11a0DEC1024HEX3f0:e7a72574f8782ae26aabDEC1520HEX5f0:b40110c4190b5622a961DEC1536HEX600:ff <a0< td="">3d1bc03cd33d70dff9DEC2032HEX7f0:d03d1bc03cd33d70dff9DEC2048HEX800:8a44126411eaa78bd51eDEC<</a0<>	DEC 16 HEX 10: 52 48 c4 95 90 14 12 6a 6e 8a 84 DEC 240 HEX f0: 06 59 02 e4 b6 20 f6 cc 36 c8 58 DEC 256 HEX 100: d3 9d 56 6b c6 bc e3 01 07 68 15 DEC 496 HEX 1f0: b6 d1 e6 c4 a5 e4 77 1c ad 79 53 DEC 512 HEX 200: c6 8c 1d 5c 55 9a 97 41 23 df 1d DEC 752 HEX 2f0: c5 ec f8 8d e8 97 fd 57 fe d3 01 DEC 768 HEX 300: ec cb e1 3d e1 fc c9 1c 11 a0 <

9 cd e4 1b 97 1 1d 1a 9e 1c f 66 43 2f 2b 5 49 f3 87 3f d f2 95 fb 11 c 52 a4 3b 89 0 1b 82 a2 59 0b c8 fa 4d С bc d6 60 65 е c a8 c5 3c 16 0 01 7e d2 97 2 ae 66 11 81 d 71 96 3e bd 7 a8 87 9b f5 4 6c 46 15 a3 ee 53 76 23 9 e 5e 57 2b f8 1 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.)

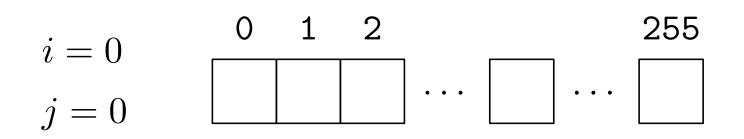


There is a bias towards 0 in the second byte output by RC4

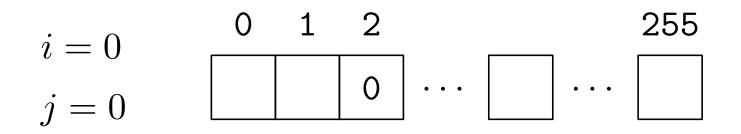
Expected: $\frac{1}{256} \approx 0.0039$

(about twice as likely to be 0)

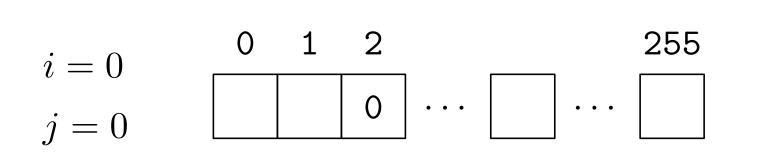
- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \dots, 255\}$



- 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. Assume that $S[1] \neq 2$ (happens with probability $\approx \frac{254}{255} \approx 1$)



- 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. Assume that $S[1] \neq 2$ (happens with probability $\approx \frac{254}{255} \approx 1$)

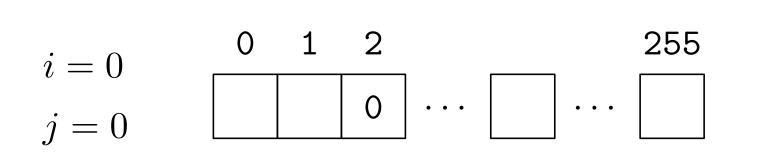


Next(st = $\langle S, i, j \rangle$):

- $i \leftarrow i+1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap S[i] and S[j]
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$

(returns a byte)

- 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. Assume that $S[1] \neq 2$ (happens with probability $\approx \frac{254}{255} \approx 1$)

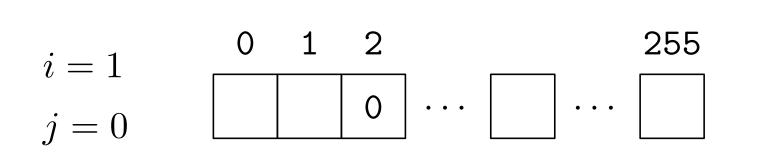


Next(st = $\langle S, i, j \rangle$ **)**:

- $i \leftarrow i+1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap S[i] and S[j]
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$

(returns a byte)

- 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. Assume that $S[1] \neq 2$ (happens with probability $\approx \frac{254}{255} \approx 1$)

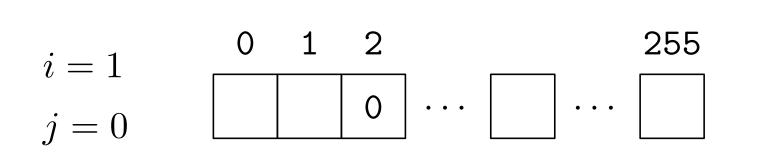


Next(st = $\langle S, i, j \rangle$ **)**:

- $i \leftarrow i+1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap S[i] and S[j]
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$

(returns a byte)

- 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. Assume that $S[1] \neq 2$ (happens with probability $\approx \frac{254}{255} \approx 1$)



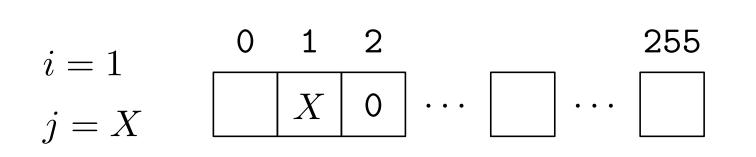
Next(st = $\langle S, i, j \rangle$ **)**:

- $i \leftarrow i+1 \pmod{256}$
 - $j \leftarrow j + S[i] \pmod{256}$
 - Swap S[i] and S[j]
 - $t = S[i] + S[j] \pmod{256}$
 - $y \leftarrow S[t]$



(returns a byte) • Return the byte y and the new state st' = $\langle S, i, j \rangle$

- 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. Assume that $S[1] \neq 2$ (happens with probability $\approx \frac{254}{255} \approx 1$)



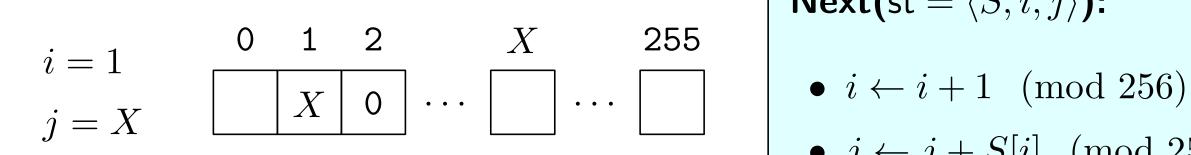
Next(st =
$$\langle S, i, j \rangle$$
):

- $i \leftarrow i+1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap S[i] and S[j]
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$



(returns a byte) • Return the byte y and the new state st' = $\langle S, i, j \rangle$

- 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. Assume that $S[1] \neq 2$ (happens with probability $\approx \frac{254}{255} \approx 1$)



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

- $j \leftarrow j + S[i] \pmod{256}$
- Swap S[i] and S[j]
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$



(returns a byte)

- 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. Assume that $S[1] \neq 2$ (happens with probability $\approx \frac{254}{255} \approx 1$)

$$i = 1$$
 0 1 2 X 255
 $j = X$ 0 1 0 ... X ...

- 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]
 - $t = S[i] + S[j] \pmod{256}$ $y \leftarrow S[t]$



(returns a byte)

- 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. Assume that $S[1] \neq 2$ (happens with probability $\approx \frac{254}{255} \approx 1$)

$$i = 1$$
 0 1 2 X 255
 $j = X$ 0 1 0 ... X ...

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 the byte y and the new state st' = $\langle S, i, j \rangle$

(returns a byte)

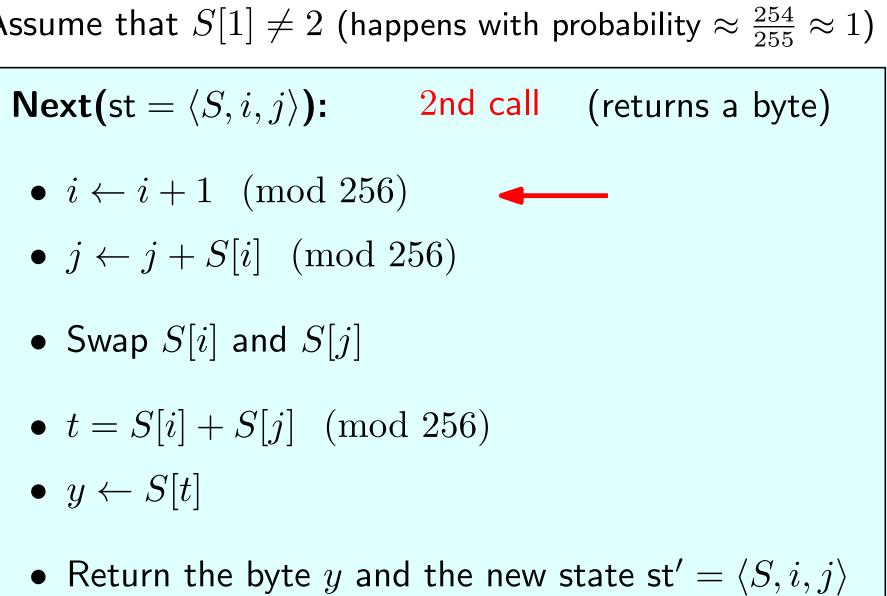
The rest of the code does not modify the state

- 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. Assume that $S[1] \neq 2$ (happens with probability $\approx \frac{254}{255} \approx 1$)

$$i = 1$$
 0 1 2 X 255
 $j = X$ 0 1 0 ... X ...

- $i \leftarrow i + 1 \pmod{256}$ $j \leftarrow j + S[i] \pmod{256}$
 - Swap S[i] and S[j]
 - $t = S[i] + S[j] \pmod{256}$ $y \leftarrow S[t]$



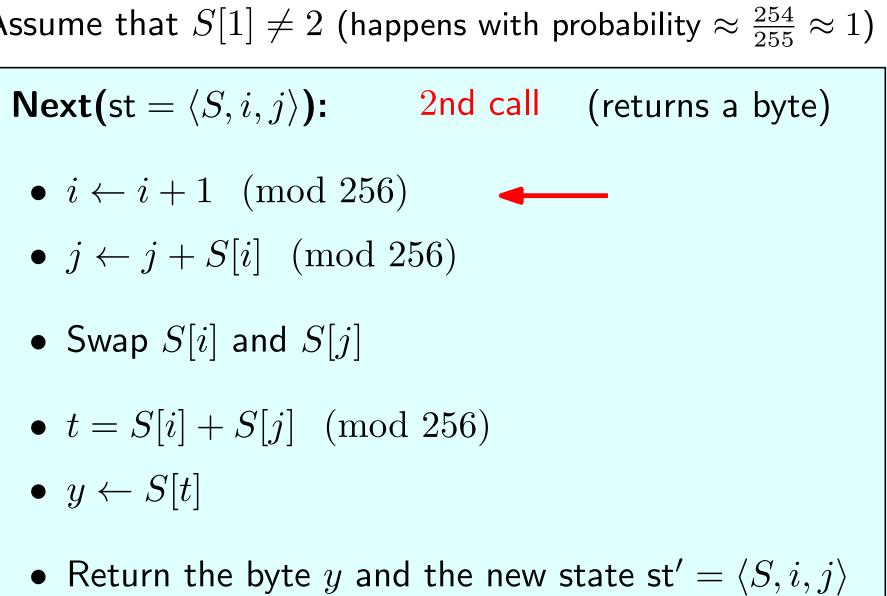


- Consider the state immediately after **Init**
- For simplicity, think of S as a uniform permutation over $\{0, 1, \ldots, 255\}$
- With probability $\approx \frac{1}{256}$ we have S[2] = 0. Assume that $S[1] \neq 2$ (happens with probability $\approx \frac{254}{255} \approx 1$)

$$i = 2$$
 0 1 2 X 255
 $j = X$ 0 1 0 ... X ...

- $i \leftarrow i + 1 \pmod{256}$ $j \leftarrow j + S[i] \pmod{256}$
 - Swap S[i] and S[j]
 - $t = S[i] + S[j] \pmod{256}$ $y \leftarrow S[t]$





- 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. Assume that $S[1] \neq 2$ (happens with probability $\approx \frac{254}{255} \approx 1$)

$$i = 2$$
 0 1 2 X 255
 $j = X$ 0 1 0 ... X ...

- $i \leftarrow i + 1 \pmod{256}$ $j \leftarrow j + S[i] \pmod{256}$
- Swap S[i] and S[j]
- $t = S[i] + S[j] \pmod{256}$ $y \leftarrow S[t]$



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



- 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. Assume that $S[1] \neq 2$ (happens with probability $\approx \frac{254}{255} \approx 1$)

$$i = 2$$
 0 1 2 X 255
 $j = X$ 0 1 0 ... X ...

- $i \leftarrow i + 1 \pmod{256}$ $j \leftarrow j + S[i] \pmod{256}$
- Swap S[i] and S[j]
- $t = S[i] + S[j] \pmod{256}$ $y \leftarrow S[t]$



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

- 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. Assume that $S[1] \neq 2$ (happens with probability $\approx \frac{254}{255} \approx 1$)

- $i \leftarrow i + 1 \pmod{256}$ $j \leftarrow j + S[i] \pmod{256}$
- Swap S[i] and S[j]
- $t = S[i] + S[j] \pmod{256}$ $y \leftarrow S[t]$



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

- 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. Assume that $S[1] \neq 2$ (happens with probability $\approx \frac{254}{255} \approx 1$)

- $i \leftarrow i + 1 \pmod{256}$ $j \leftarrow j + S[i] \pmod{256}$
 - Swap S[i] and S[j]
 - $t = S[i] + S[j] \pmod{256}$ $y \leftarrow S[t]$

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

- 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. Assume that $S[1] \neq 2$ (happens with probability $\approx \frac{254}{255} \approx 1$)

- $i \leftarrow i + 1 \pmod{256}$ $j \leftarrow j + S[i] \pmod{256}$
 - Swap S[i] and S[j]
 - $t = S[i] + S[j] \pmod{256}$ $y \leftarrow S[t]$

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



- 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. Assume that $S[1] \neq 2$ (happens with probability $\approx \frac{254}{255} \approx 1$)

$$i = 2$$
 0 1 2 X 255
 $j = X$ 0 1 2 ... 0 ...

$$t = X$$
 Output byte $y = 0$

- $i \leftarrow i + 1 \pmod{256}$ $j \leftarrow j + S[i] \pmod{256}$
 - Swap S[i] and S[j]
 - $t = S[i] + S[j] \pmod{256}$ $y \leftarrow S[t]$

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



- 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. Assume that $S[1] \neq 2$ (happens with probability $\approx \frac{254}{255} \approx 1$)

$$i = 2$$
 0 1 2 X 255
 $j = X$ 0 1 2 ... 0 ...

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 = $\langle S, i, j \rangle$):

- $i \leftarrow i+1 \pmod{256}$ $j \leftarrow j+S[i] \pmod{256}$
- Swap S[i] and S[j]
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$

(returns a byte)

- 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. Assume that $S[1] \neq 2$ (happens with probability $\approx \frac{254}{255} \approx 1$)

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 = $\langle S, i, j \rangle$):

- $i \leftarrow i+1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap S[i] and S[j]
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$

(returns a byte)

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

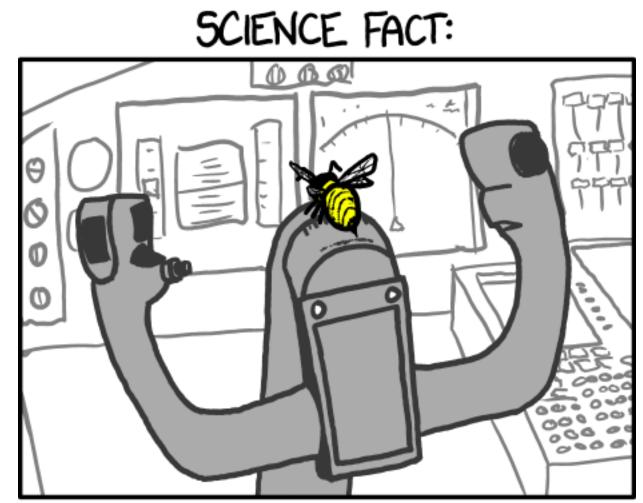
- The output bias is indicative of structural problems with RC4
- Other biases have been found in other bytes of the RC4 state
- Severe enough to allow recovery of plaintext from ciphertext when RC4 is used for encryption!



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



PHYSICISTS STILL CAN'T EXPLAIN HOW BUMBLEBEES CAN FLY AIRPLANES.

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

$$k = \mathsf{IV} \parallel k'$$

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

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



• 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

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



- 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

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

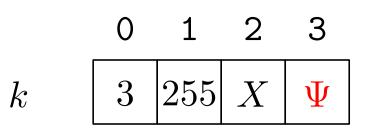


- 3-byte IV, 13 bytes key
- Key recovery attack!
- We show a simplified attack that recovers the first byte of the key (i.e., k[3])

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

- 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)

- 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



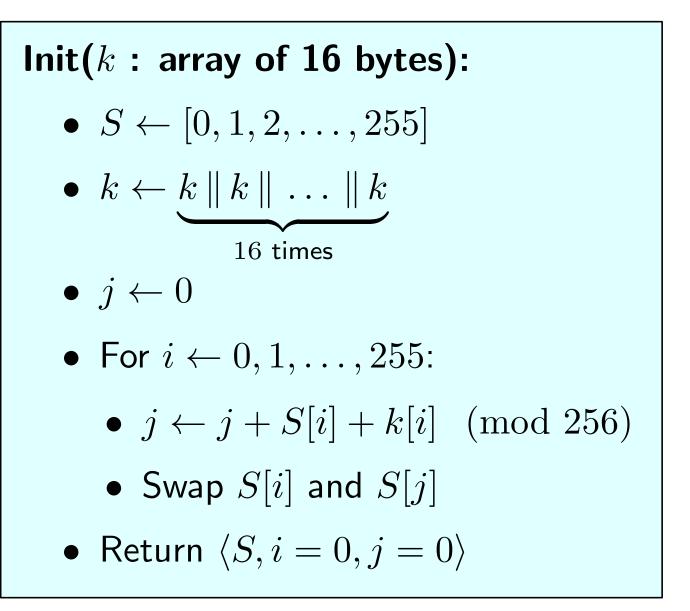
• • •

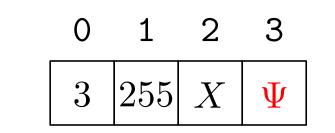
(attacks for other combinations are also known)

k

- 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 $\frac{1}{256^2} = \frac{1}{65536}$

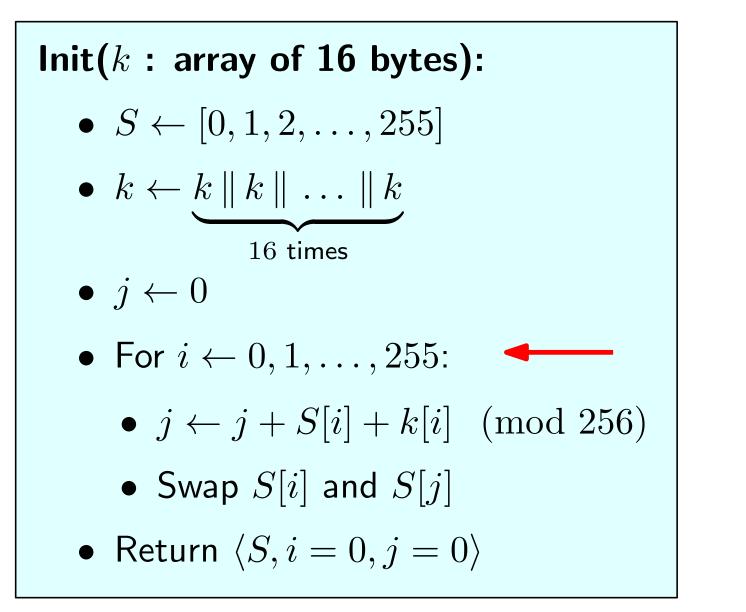
this is just one possibility (attacks for other combinations are also known)

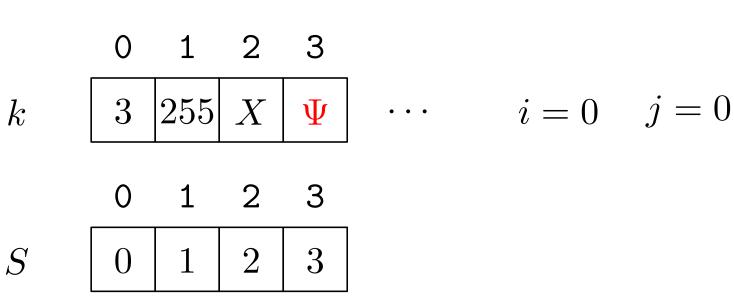




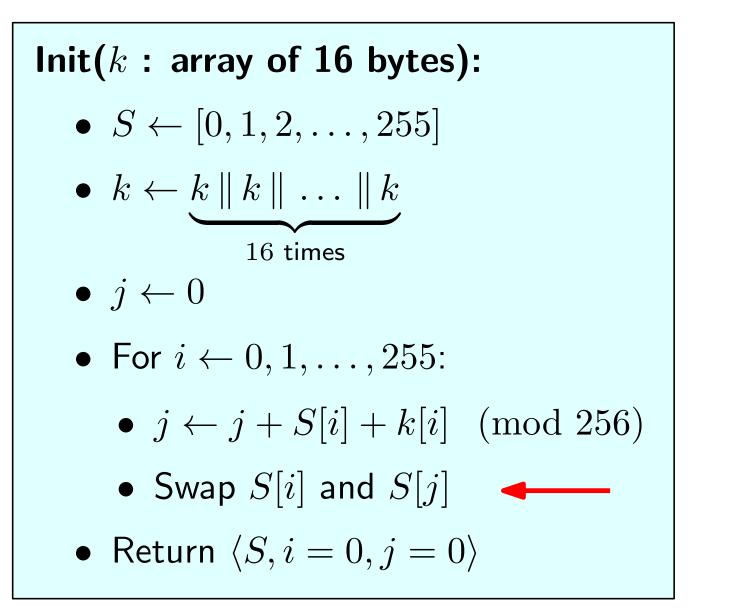
• • •

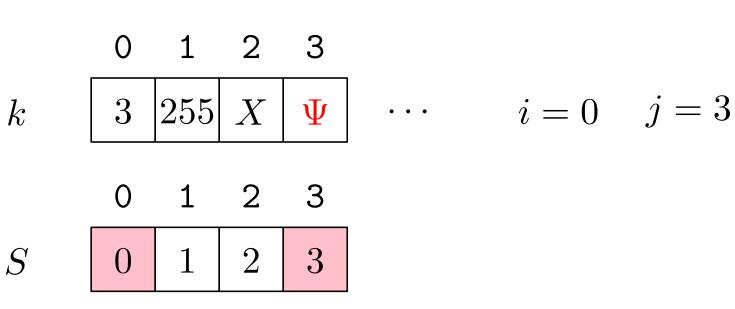
- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$



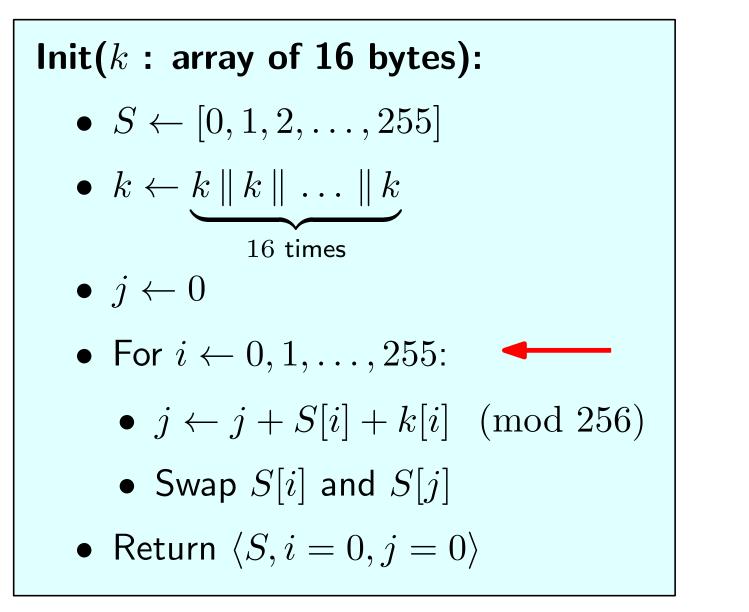


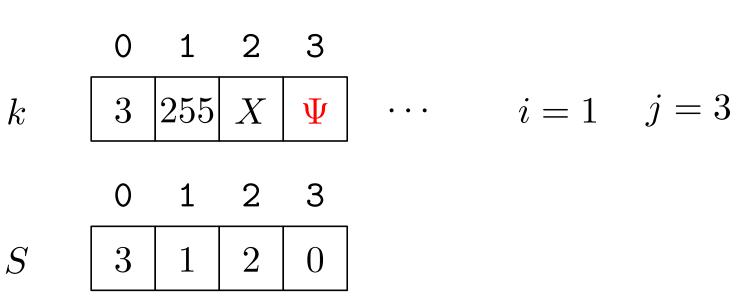
- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$



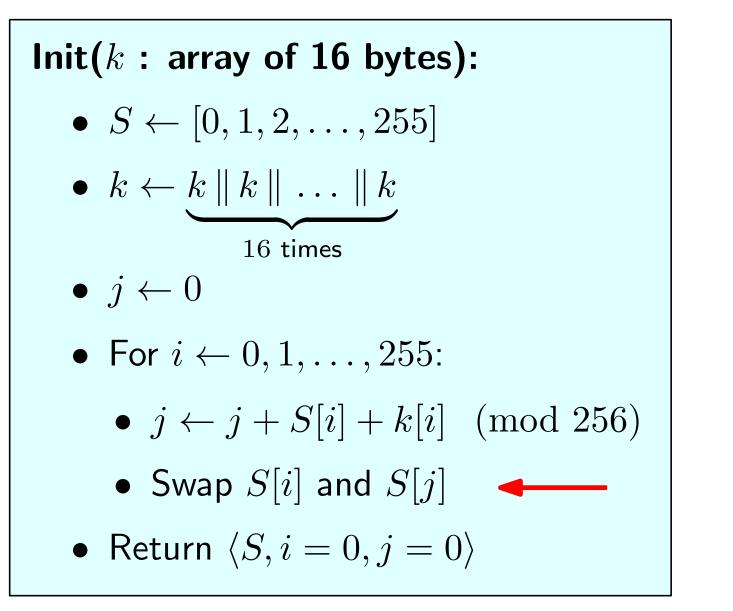


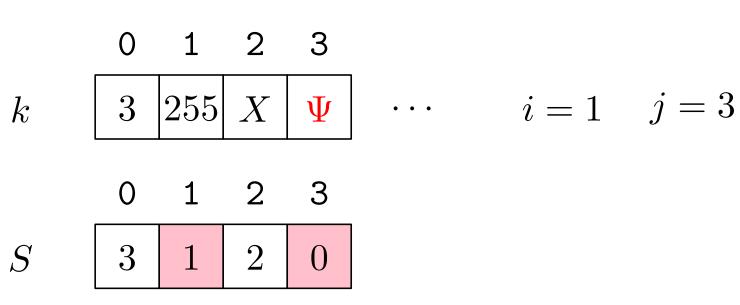
- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$



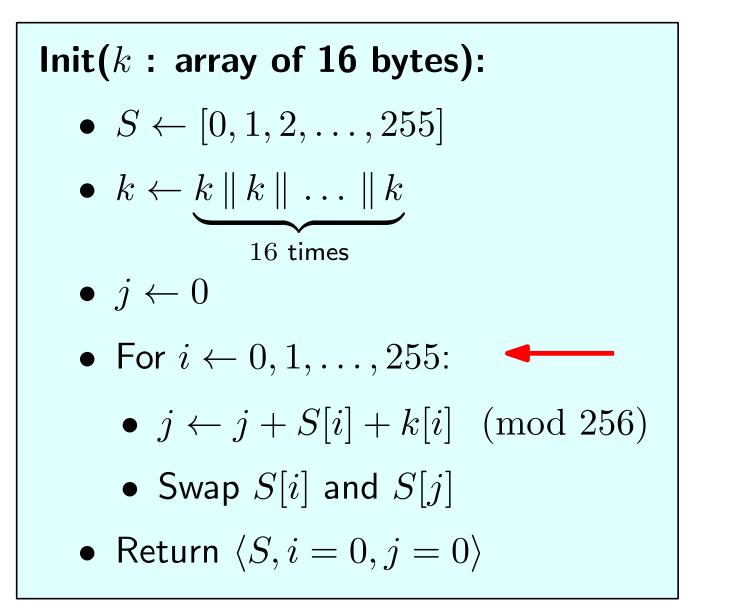


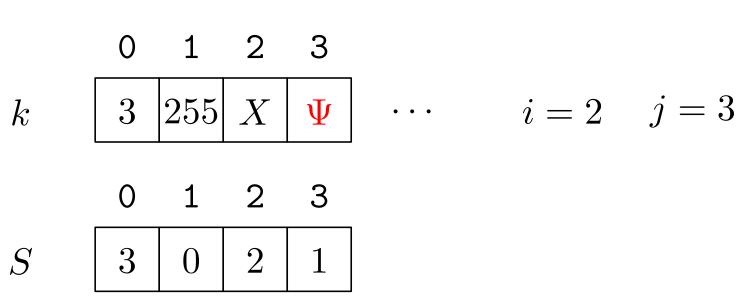
- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$





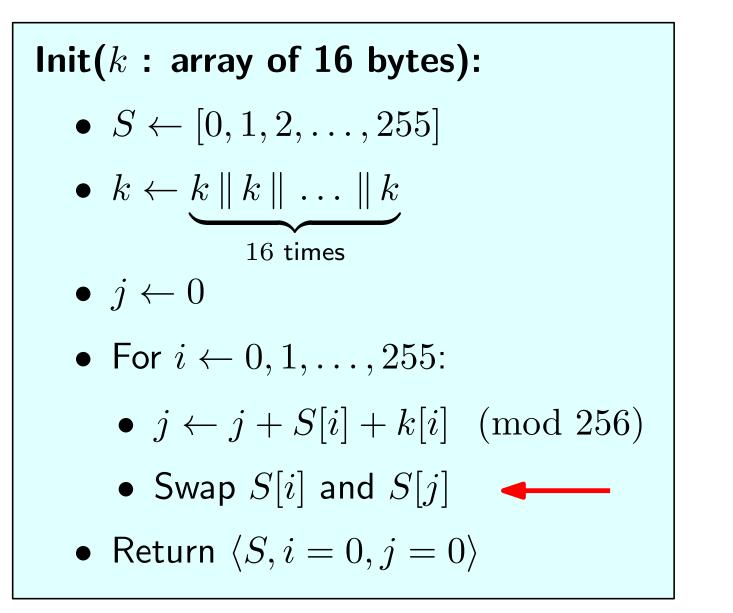
- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

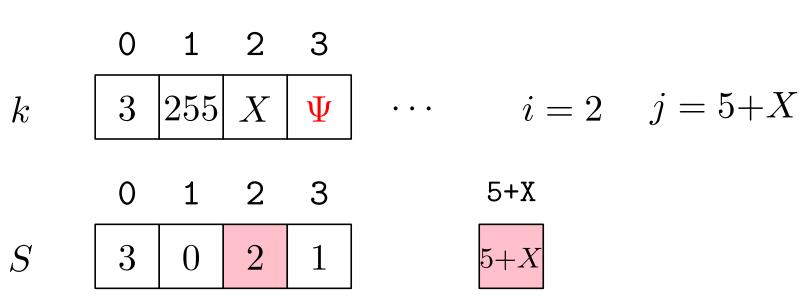




- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

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



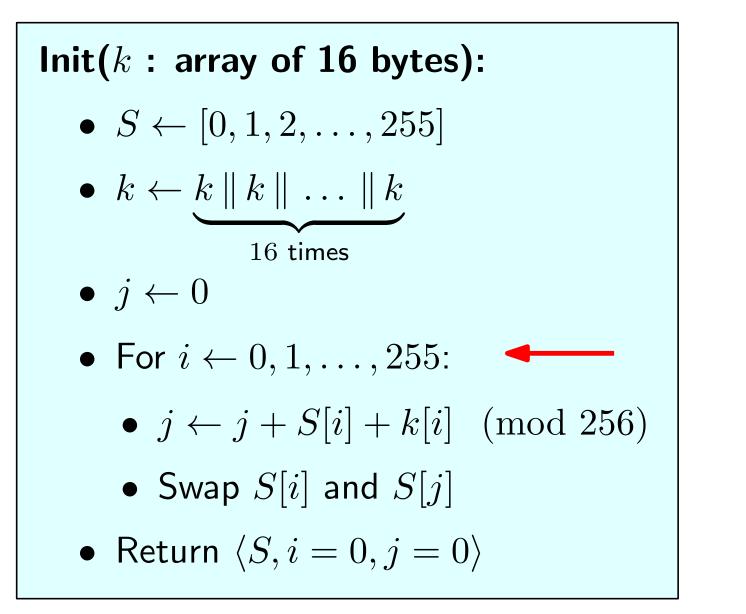


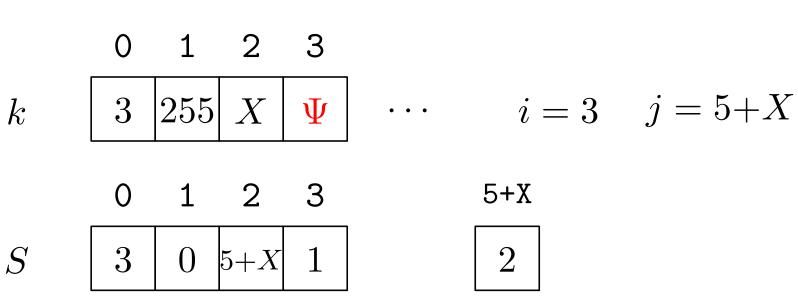
5+X



- 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)



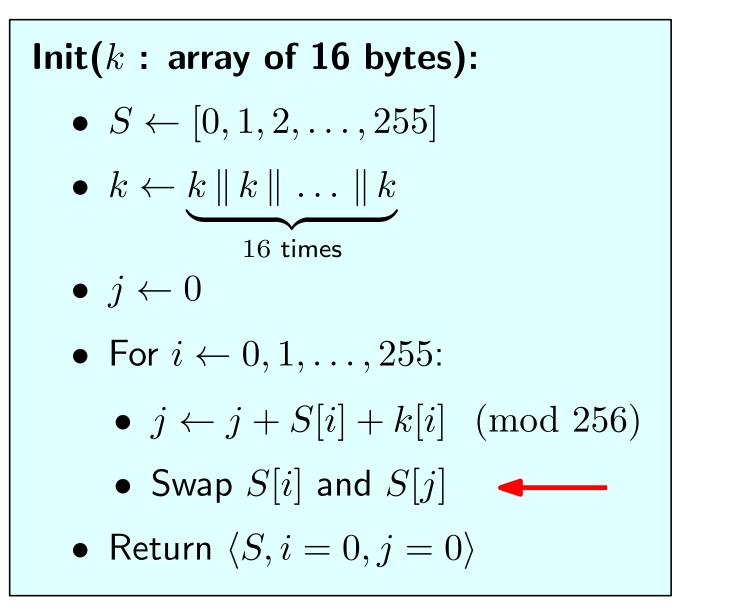


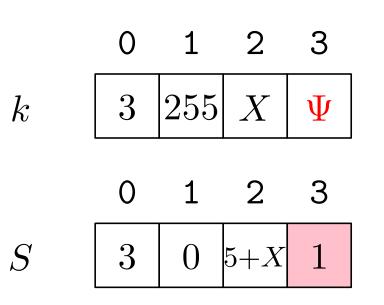
5+X



- 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)





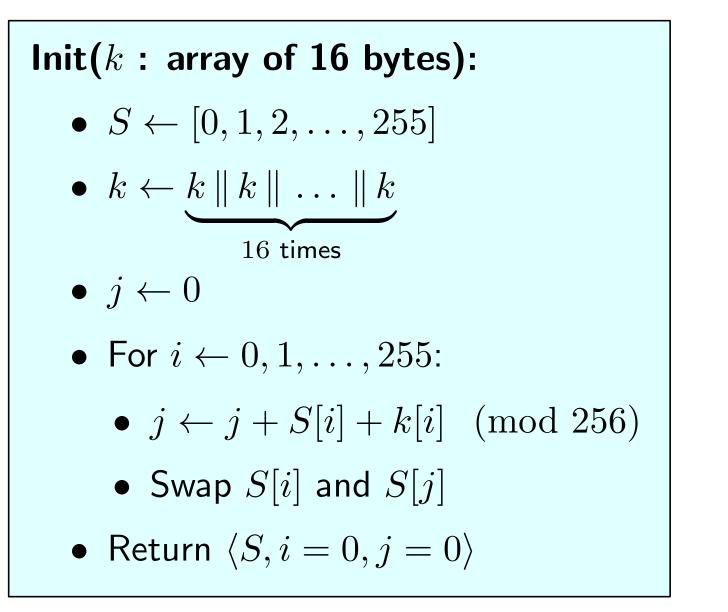


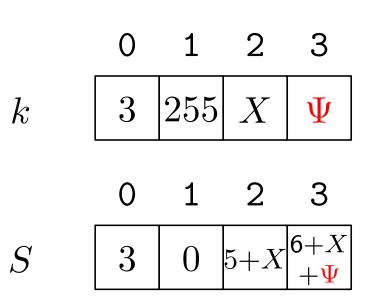
5+X

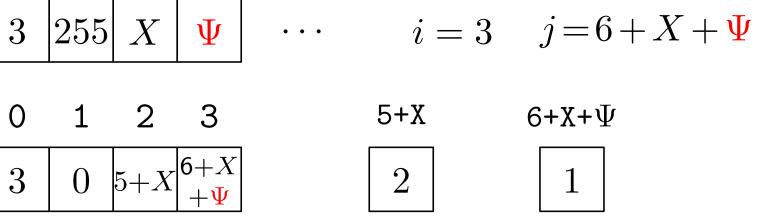
 $6+X+\Psi$

9	6+X
Ζ	$+\Psi$

- 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 $\frac{1}{256^2} = \frac{1}{65536}$

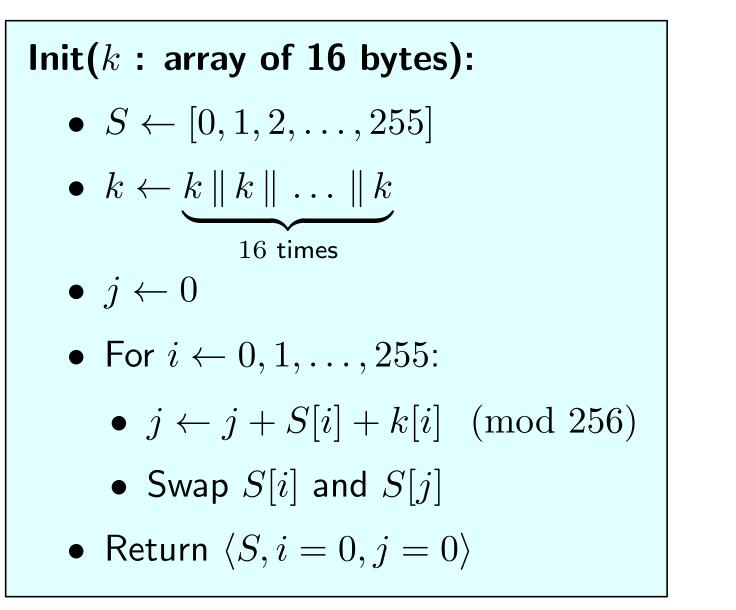


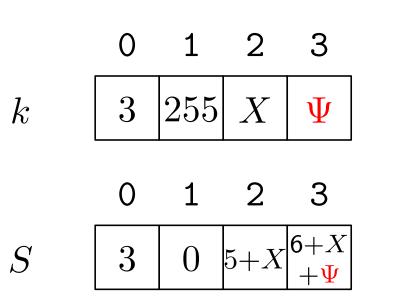




- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

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





With probability $\approx 5\%$, S[0], S[1], and S[3] are not modified in the remaining iterations of Init

$3 |255| X | \Psi | \cdots \qquad i = 3 \qquad j = 6 + X + \Psi$

5+X	$6+X+\Psi$	
2	1	

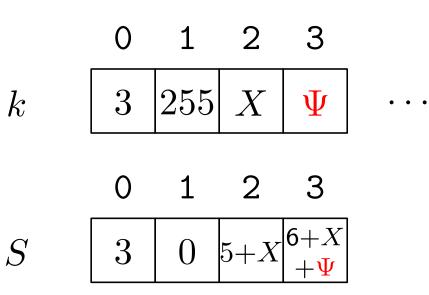
- Recall that **IV**s are not kept secret!
- The adversary waits until the IV takes the form $\langle 3, 255, X \rangle$ (for some value X)
- Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility

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

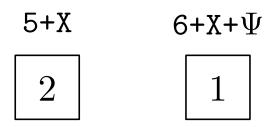
- $i \leftarrow i+1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap S[i] and S[j]
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$

• Return
$$y$$
 and $st' = \langle S, i, j \rangle$



With probability $\approx 5\%$, S[0], S[1], and S[3] are not modified in the remaining iterations of Init

(attacks for other combinations are also known)



- 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)

i = 1

• Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility

 Ψ

3

. . .

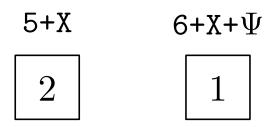
Next(st = $\langle S, i, j \rangle$):

- $i \leftarrow i+1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap S[i] and S[j]
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$

• Return
$$y$$
 and $st' = \langle S, i, j \rangle$

$$k = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 3 & 255 & X & \Psi \\ & 0 & 1 & 2 & 3 \\ S & 3 & 0 & 5+X \begin{bmatrix} 6+X \\ +\Psi \end{bmatrix}$$

(attacks for other combinations are also known)



With probability $\approx 5\%$, S[0], S[1], and S[3] are not modified in the remaining iterations of Init

- 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)

i = 1

j = 0

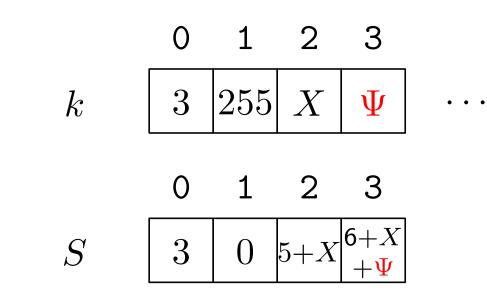
• Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility

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

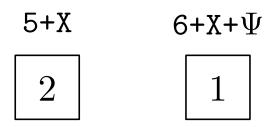
- $i \leftarrow i+1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap S[i] and S[j]
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$

• Return
$$y$$
 and $st' = \langle S, i, j \rangle$



With probability $\approx 5\%$, S[0], S[1], and S[3] are not modified in the remaining iterations of Init

(attacks for other combinations are also known)



k

- 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)

i = 1

j = 0

t = 3

• Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility

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

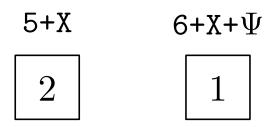
- $i \leftarrow i+1 \pmod{256}$
- $j \leftarrow j + S[i] \pmod{256}$
- Swap S[i] and S[j]
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$

• Return
$$y$$
 and $st' = \langle S, i, j \rangle$

0 1 2 3 3 |255| X | Ψ 0 1 2 3 $0 |_{5+X}|^{6+X}$ S3

• • •

(attacks for other combinations are also known)



With probability $\approx 5\%$, S[0], S[1], and S[3] are not modified in the remaining iterations of Init

k

- 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)

i = 1

j = 0

t = 3

y = S[3]

• Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility

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]
- $t = S[i] + S[j] \pmod{256}$
- $y \leftarrow S[t]$

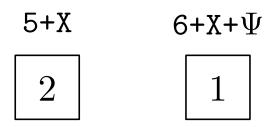
• Return
$$y$$
 and $st' = \langle S, i, j \rangle$

$$k = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 3 & 255 & X & \Psi \end{bmatrix}$$

$$k = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \\ 3 & 0 & 5 + X \begin{bmatrix} 6 + X \\ + \Psi \end{bmatrix}$$

• • •

(attacks for other combinations are also known)



With probability $\approx 5\%$, S[0], S[1], and S[3] are not modified in the remaining iterations of Init

- 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)

i = 1

j = 0

 $\begin{array}{c} t = 3 \\ y = S[3] \end{array}$

• Happens with probability $\frac{1}{256^2} = \frac{1}{65536}$

this is just one possibility

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' = $\langle S, i, j \rangle$

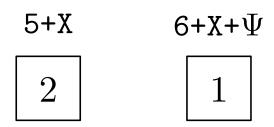
$$k = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 3 & 255 & X & \Psi \end{bmatrix}$$

$$k = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \\ \hline 3 & 0 & 5 + X \begin{bmatrix} 6 + X \\ + \Psi \end{bmatrix}$$

What's the first byte output by Next (when i = j = 0)? $6 + X + \Psi$

• • •

(attacks for other combinations are also known)



With probability $\approx 5\%$, S[0], S[1], and S[3] are not modified in the remaining iterations of Init

- 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

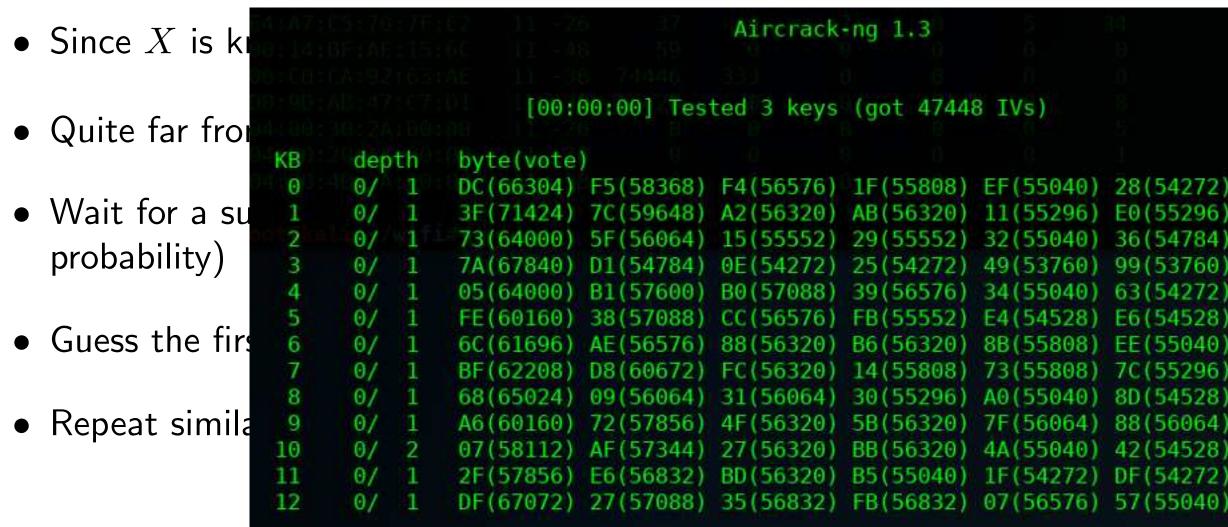
- 5% of the time the adversary sees $6 + X + \Psi$
- Since X is known (it is part of the IV), the adversary can recover Ψ
- Quite far from uniform: $\frac{1}{256} \approx 0.4\%$

- 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)

- 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



• 5% of the time the adversary sees $6 + X + \Psi$



KEY FOUND! [DC:3F:73:7A:05:FE:6C:BF:68:A6:6B:2F:DF] Decrypted correctly: 100%

E0(55296) 36(54784) 99(53760) 63(54272) E6(54528) EE(55040) 7C(55296) 8D(54528) 88(56064) 42(54528) DF(54272)

leaked (with some

is reconstructed

Introduced in 2008. Secure replacement for RC4

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



Daniel J. Bernstein

Introduced in 2008. Secure replacement for RC4

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

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



Daniel J. Bernstein

Introduced in 2008. Secure replacement for RC4

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

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

The core of ChaCha20 is a fixed permutation $P: \{0,1\}^{512} \rightarrow \{0,1\}^{512}$ 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

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



Daniel J. Bernstein



Introduced in 2008. Secure replacement for RC4

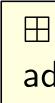
Takes a 256-bit key k and a 64-bit IV

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

The core of ChaCha20 is a fixed permutation $P: \{0,1\}^{512} \rightarrow \{0,1\}^{512}$ 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

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





Daniel J. Bernstein

H denotes word-wise modular addition (of 32-bit words)

Introduced in 2008. Secure replacement for RC4

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

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

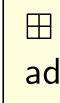
The core of ChaCha20 is a fixed permutation $P: \{0,1\}^{512} \rightarrow \{0,1\}^{512}$ on 512-bit strings

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

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

Output stream:

 $F_k(\mathsf{IV} \parallel \langle 0 \rangle), F_k(\mathsf{IV} \parallel \langle 1 \rangle), F_k(\mathsf{IV} \parallel \langle 2 \rangle), \ldots$







Daniel J. Bernstein

H denotes word-wise modular addition (of 32-bit words)

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

Introduced in 2008. Secure replacement for RC4

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

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

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

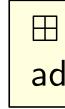
The permutation P is used to construct a keyed function with a 256-bit key, 128-bit inputs and 512-bit outputs

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

Output stream:

 $F_k(\mathsf{IV} \parallel \langle 0 \rangle), F_k(\mathsf{IV} \parallel \langle 1 \rangle), F_k(\mathsf{IV} \parallel \langle 2 \rangle), \ldots$

Not patented. Several public domain implementations available







Daniel J. Bernstein

H denotes word-wise modular addition (of 32-bit words)

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

Block Ciphers

A block cipher is...

Block Ciphers

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

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

Block Ciphers

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

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

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

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

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

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

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

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

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

Padding (with care)

Recall that we can always build a stream cipher from a block cipher For example:

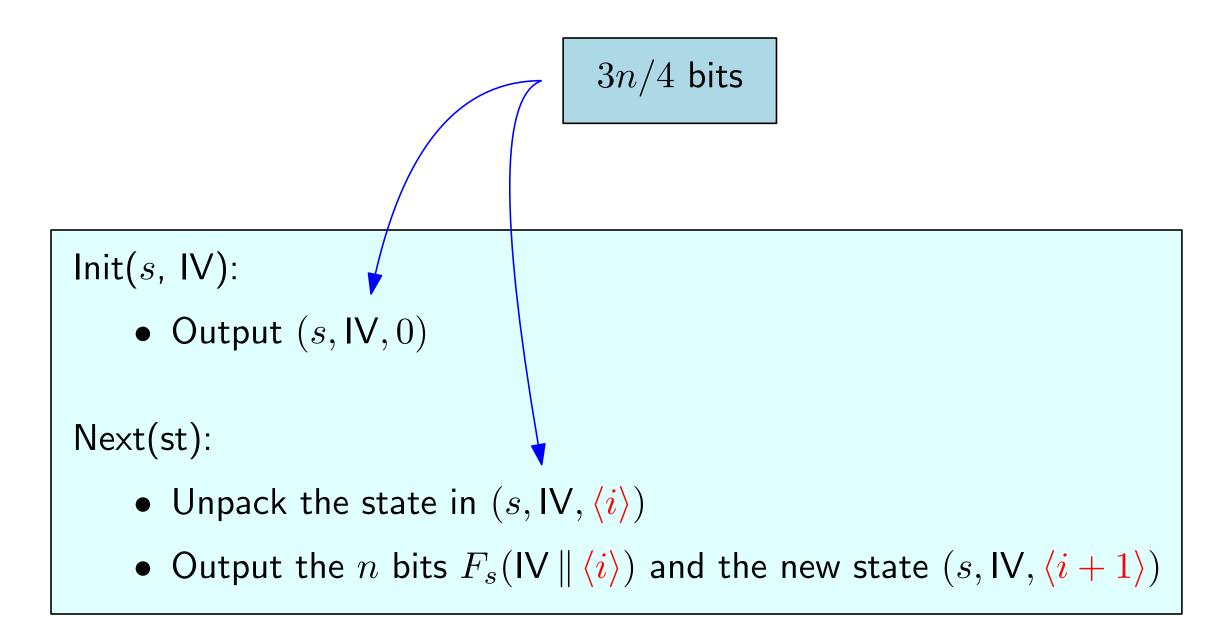
Init(s, IV):

• Output (s, IV, 0)

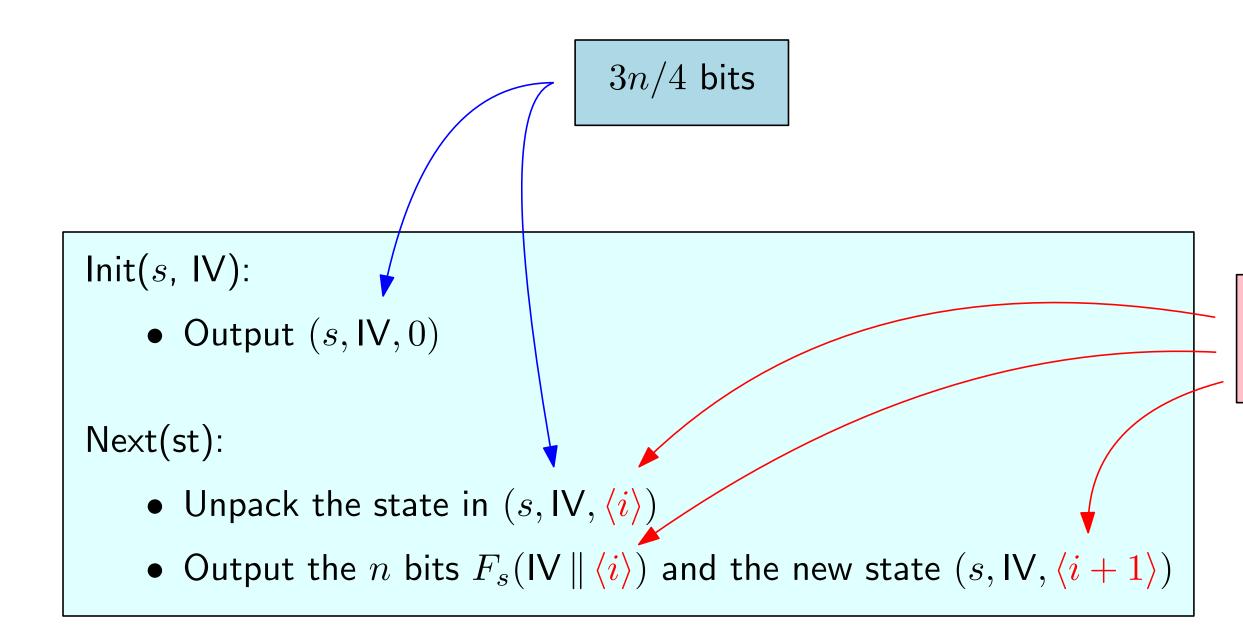
Next(st):

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

Recall that we can always build a stream cipher from a block cipher For example:



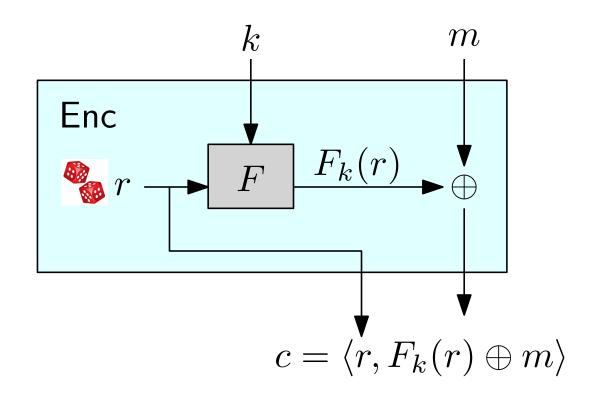
Recall that we can always build a stream cipher from a block cipher For example:



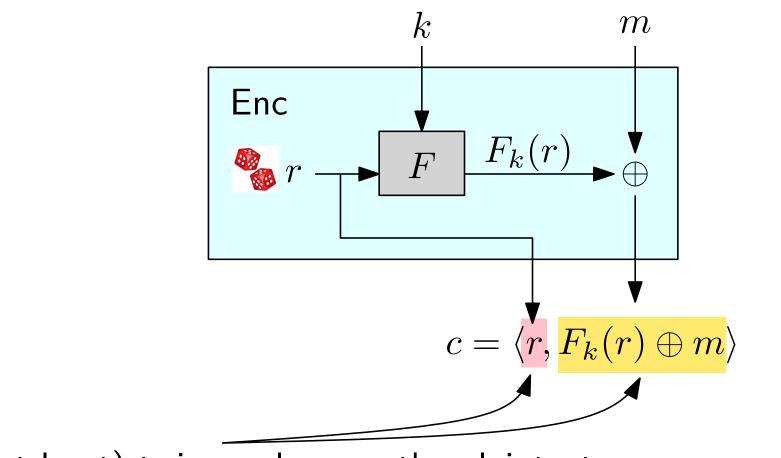
$\langle i \rangle = {\rm Binary\ encoding}$ of $i\ {\rm using\ } n/4\ {\rm bits}$

- 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*)

- 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*)

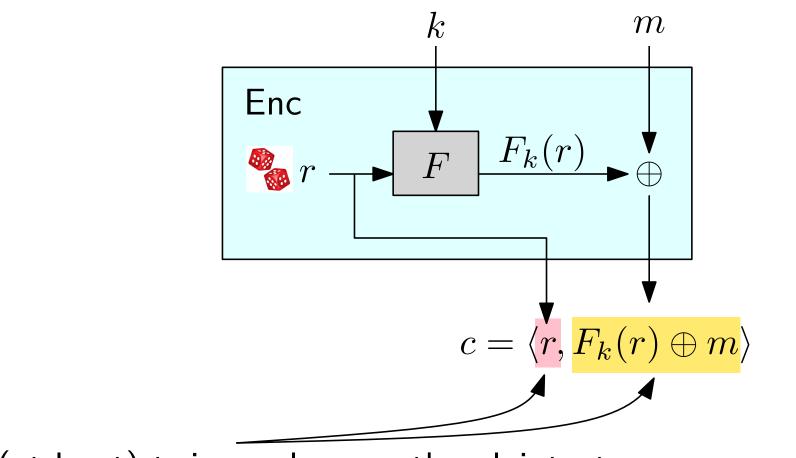


- We already have seen how to encrypt a message using a stream cipher.
- We have also seen how to encrypt a message using a block cipher (i.e., a pseudorandom) permutation*)



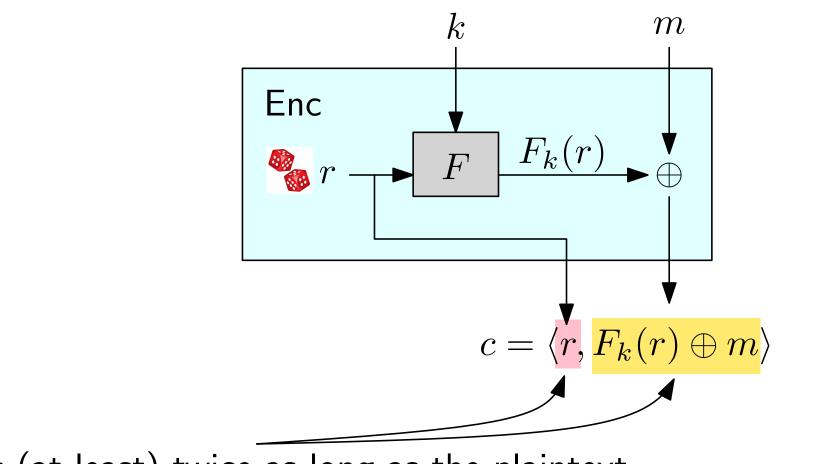
• The ciphertext is (at least) twice as long as the plaintext

- 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?

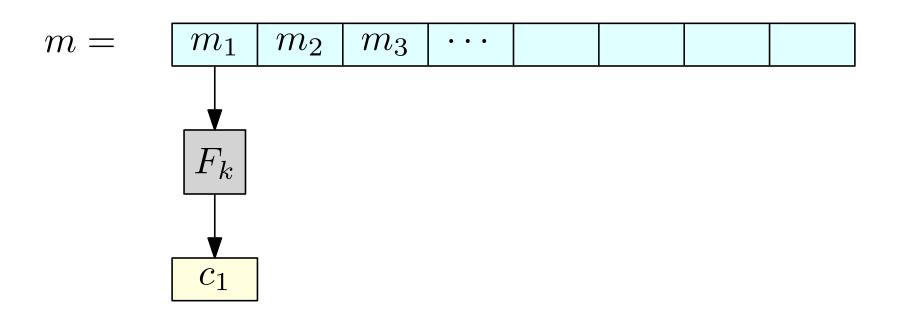
- We already have seen how to encrypt a message using a stream cipher.
- We have also seen how to encrypt a message using a block cipher (i.e., a pseudorandom) permutation*)



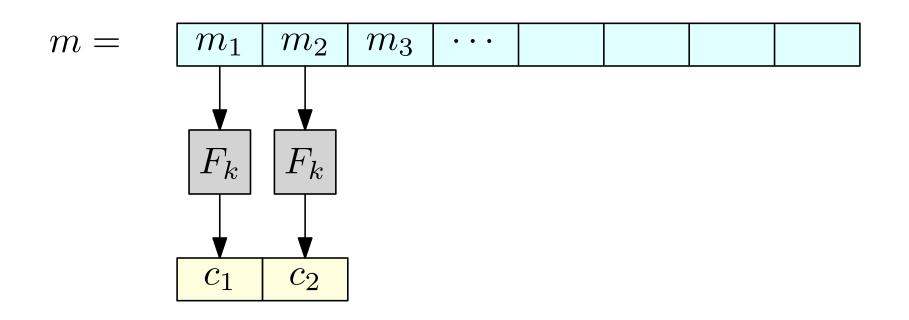
- The ciphertext is (at least) twice as long as the plaintext
- Can we do better? Several options (modes of operations)

First idea:

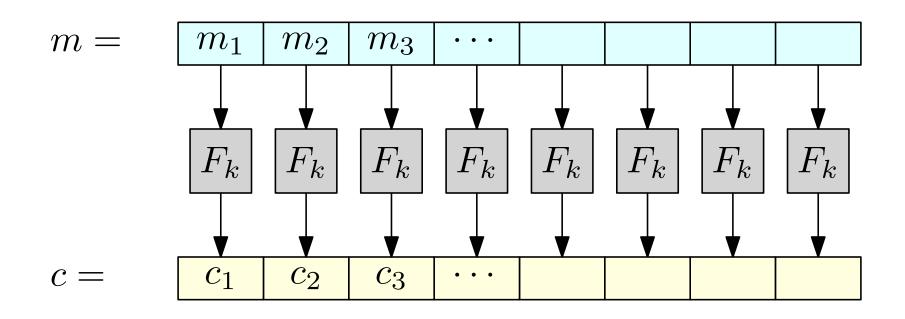
First idea:



First idea:

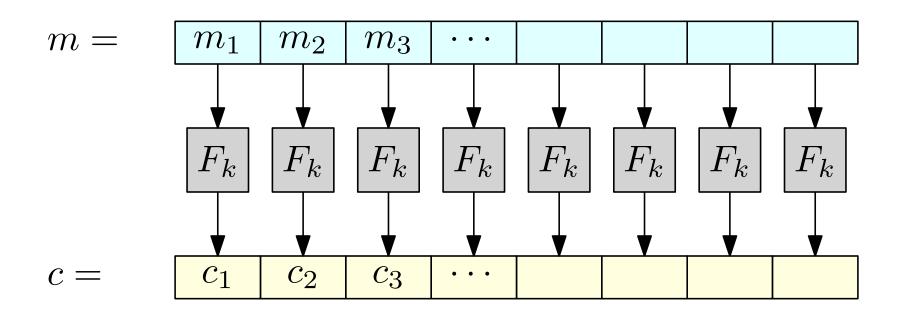


First idea:



First idea:

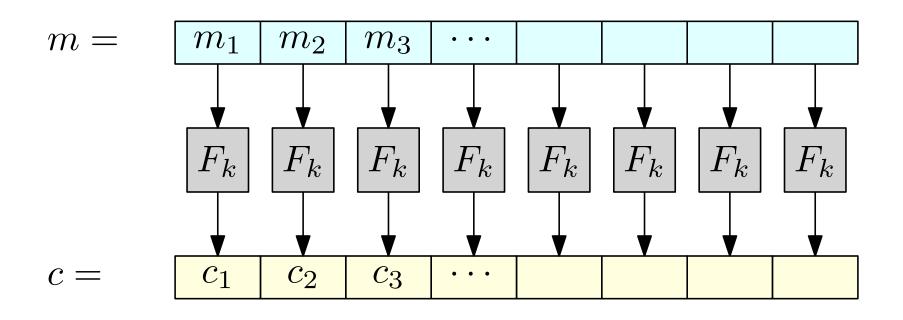
• Encrypt each block of the message independently



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

First idea:

• Encrypt each block of the message independently



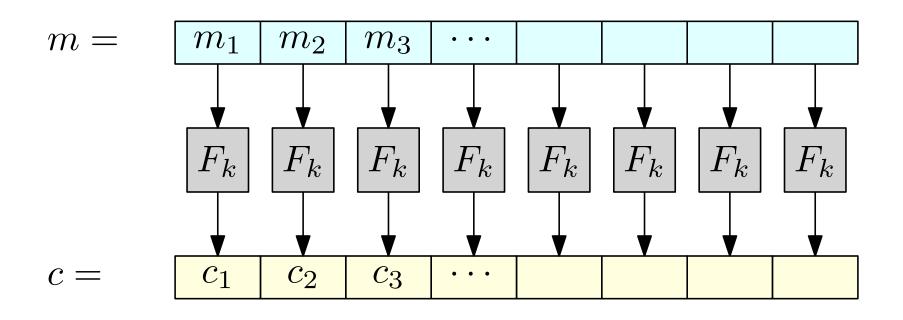
Encrypting: $c_i = F_k(m_i)$

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

• No ciphertext expansion!

First idea:

• Encrypt each block of the message independently



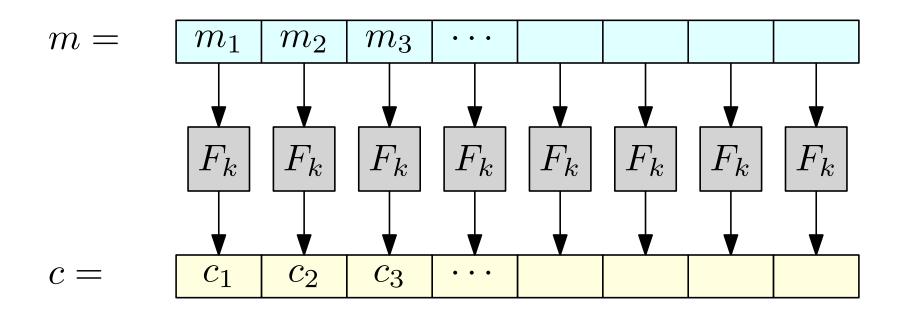
Encrypting: $c_i = F_k(m_i)$

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

- No ciphertext expansion!
- Is it CPA-secure?

First idea:

• Encrypt each block of the message independently



Encrypting: $c_i = F_k(m_i)$

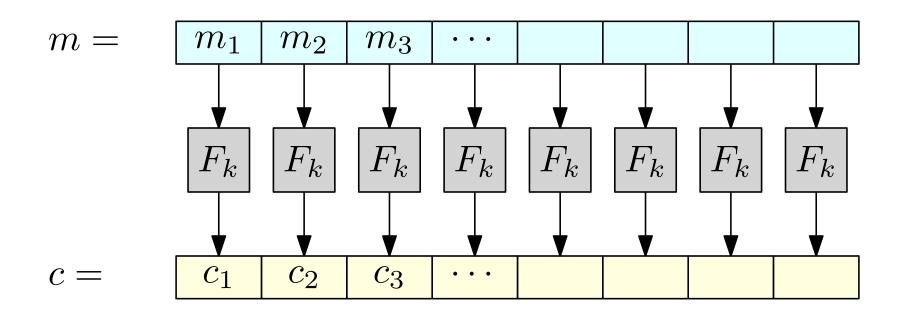


- No ciphertext expansion!
- Is it CPA-secure?

No! Encryption is deterministic!

First idea:

• Encrypt each block of the message independently



Encrypting: $c_i = F_k(m_i)$



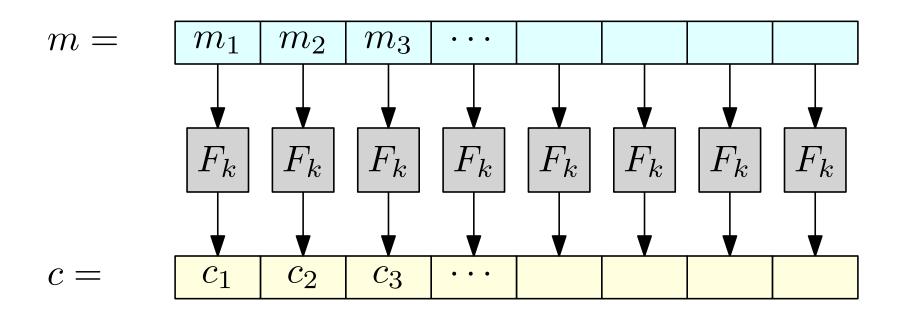
- No ciphertext expansion!
- Is it CPA-secure?

No! Encryption is deterministic!

• Is it EAV-secure?

First idea:

• Encrypt each block of the message independently



Encrypting: $c_i = F_k(m_i)$

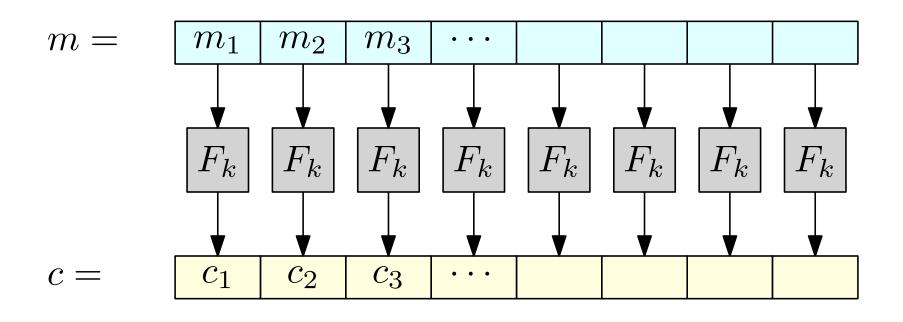
Decrypting: $m_i = F_k^{-1}(c_i)$

- No ciphertext expansion!
- Is it CPA-secure?
- Is it EAV-secure?

No! Encryption is deterministic! [Dem0]

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!

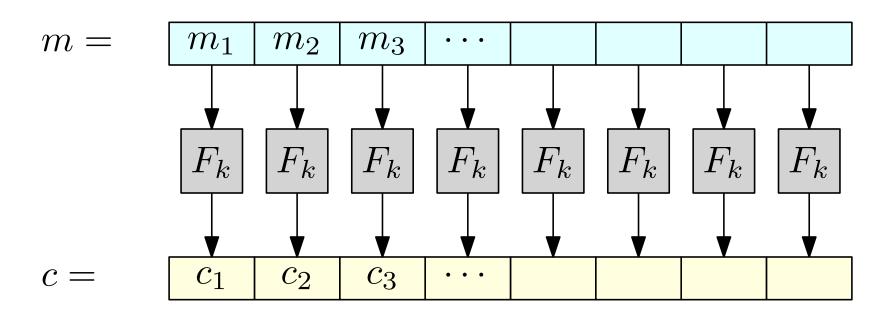
[Demo]

• Is it EAV-secure?

No! It's just a fancy substitution cipher! (Frequency analysis)

First idea:

• Encrypt each block of the message independently



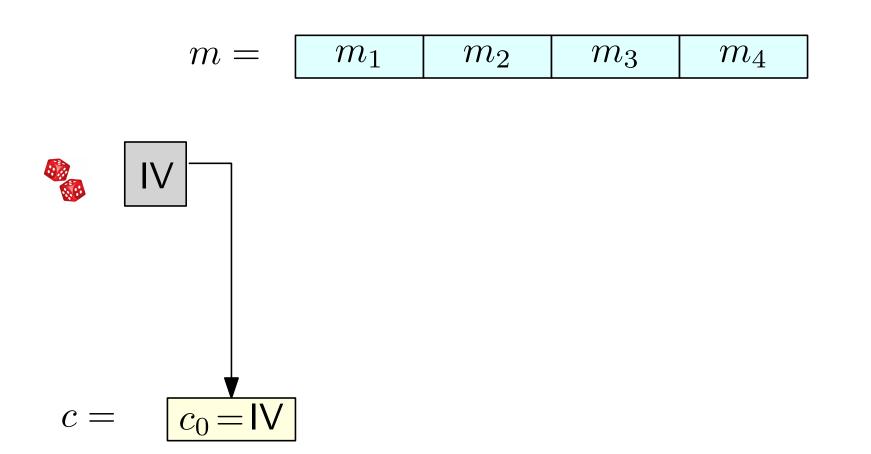
Encrypting: $c_i = F_k(m_i)$

- No ciphertext expansion!
- Is it CPA-secure?
- Is it EAV-secure?

Decrypting: $m_i = F_k^{-1}(c_i)$ No! Encryption is deterministic! [Demo]

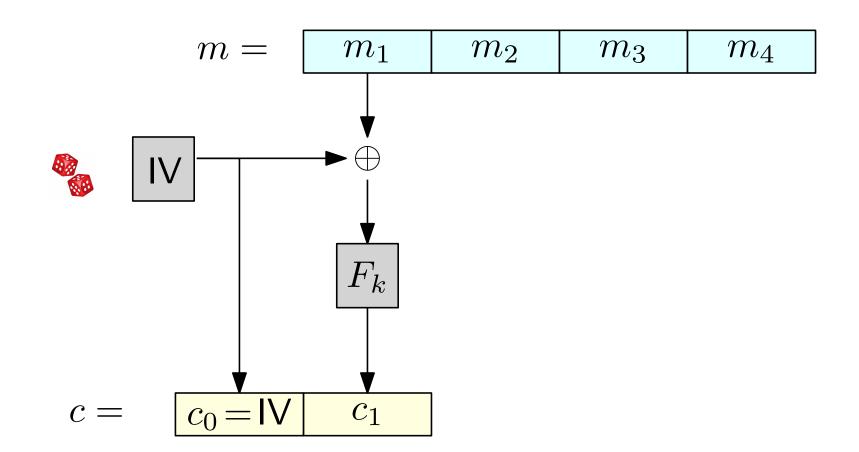


No! It's just a fancy substitution cipher! (Frequency analysis)



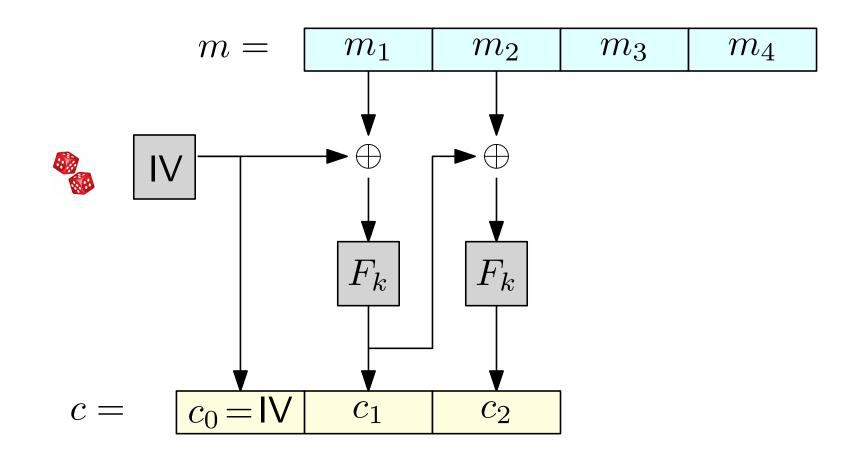
Encrypting:

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



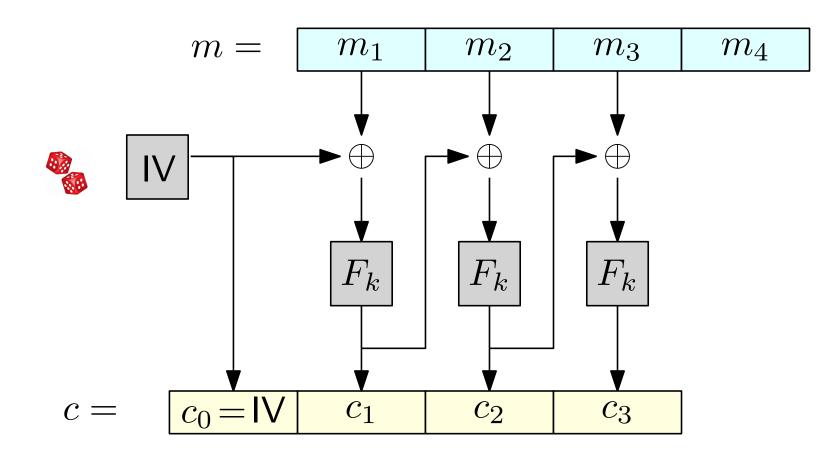
- A random IV is chosen and sent as the first block c_0 of the ciphertext
- Each block m_i of the message is XORed with the previous ciphertext block before applying F_k

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



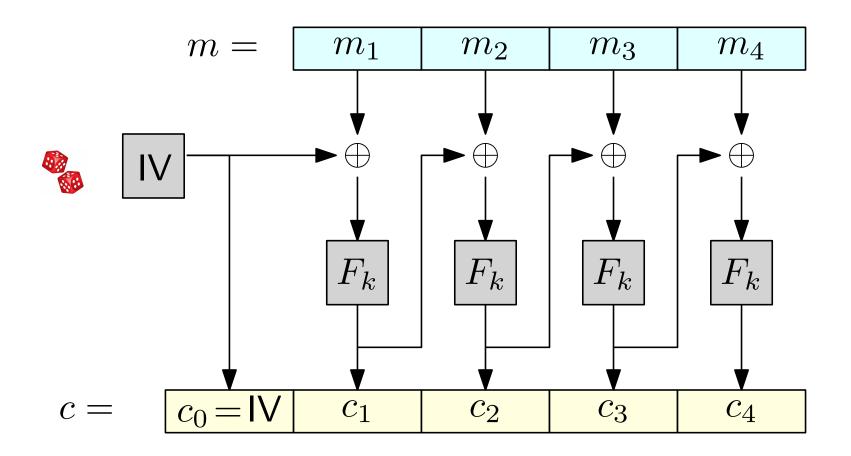
- A random IV is chosen and sent as the first block c_0 of the ciphertext
- Each block m_i of the message is XORed with the previous ciphertext block before applying F_k

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



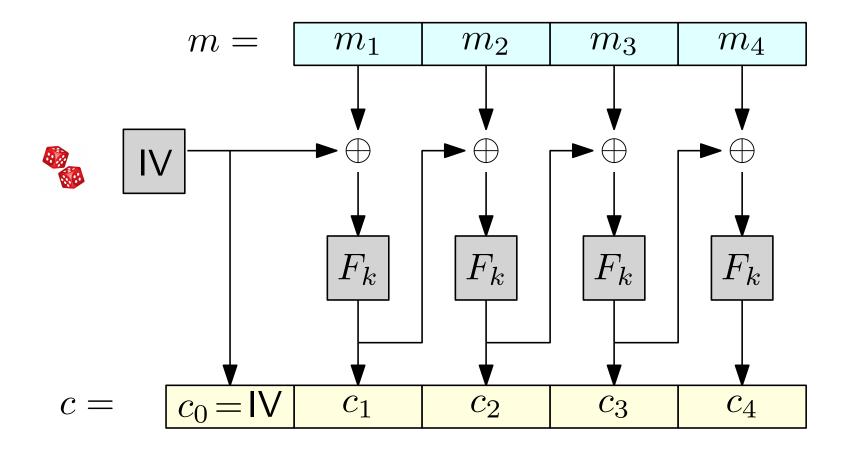
- A random IV is chosen and sent as the first block c_0 of the ciphertext
- Each block m_i of the message is XORed with the previous ciphertext block before applying F_k

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



- 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

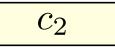
$$c_i = F_k(c_{i-1} \oplus m_i)$$



Decrypting:

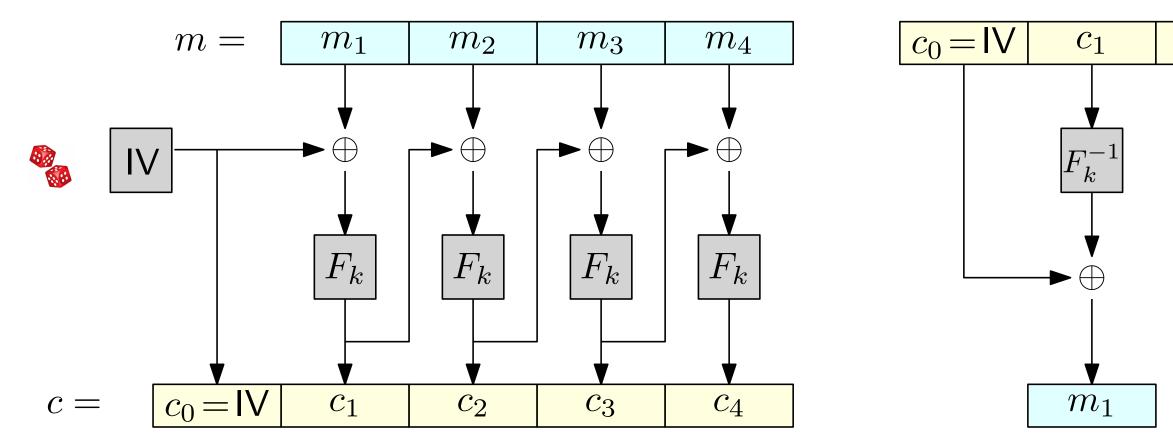
• To decrypt m_i we need c_{i-1}





 $c_0 = \mathsf{IV}$

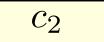
 c_1

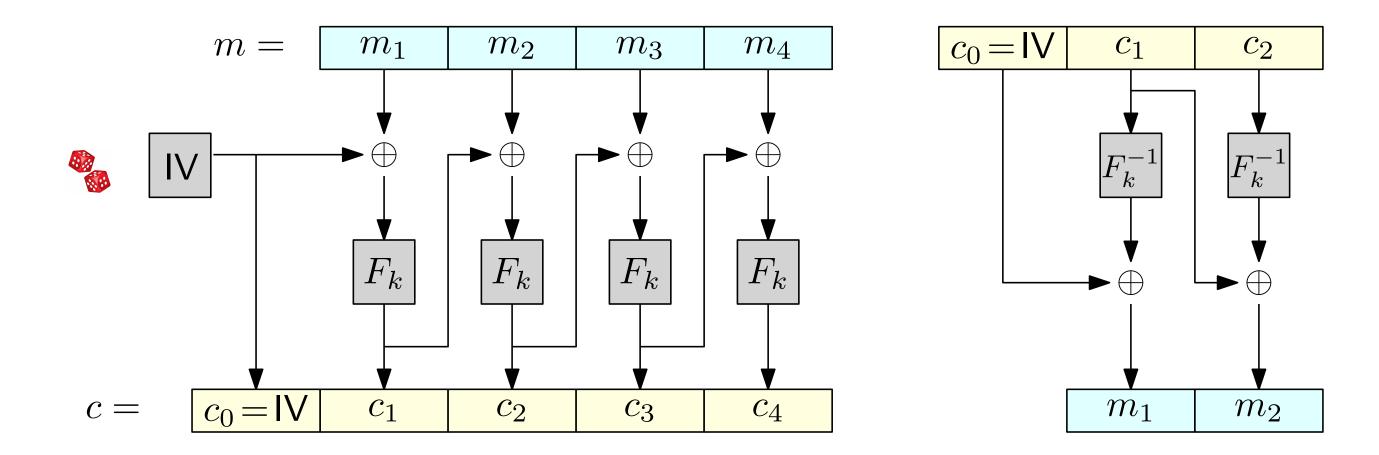


Decrypting:

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

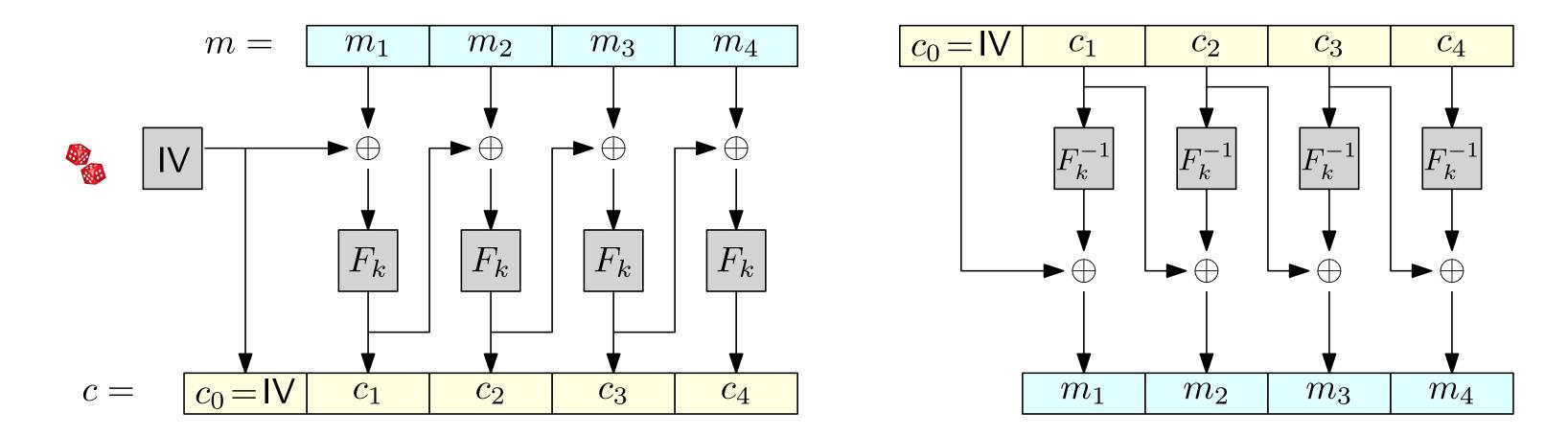






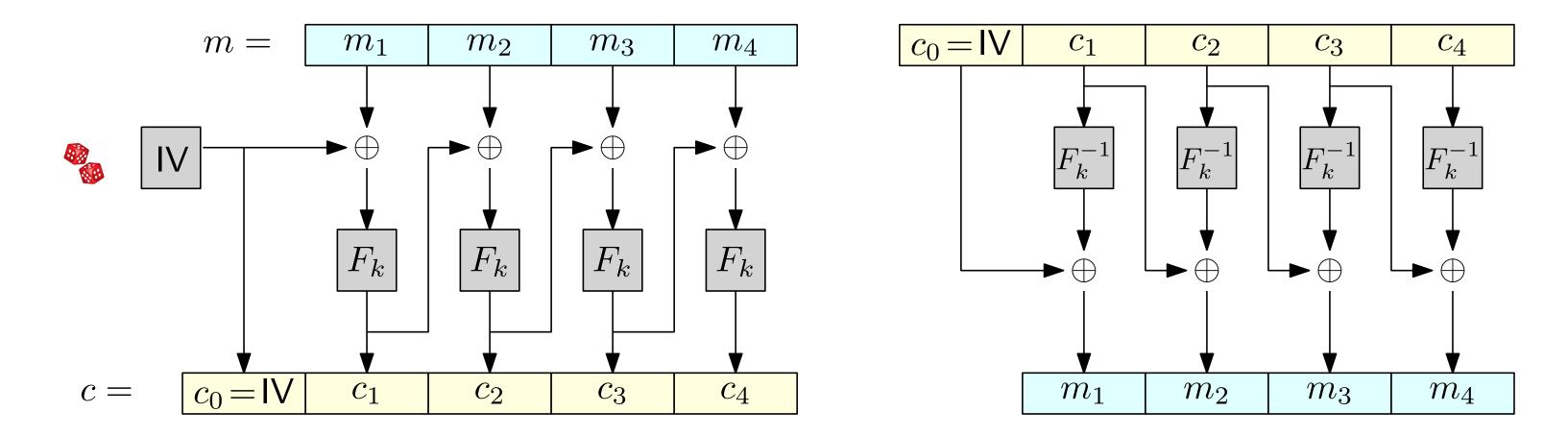
Decrypting:

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



Decrypting:

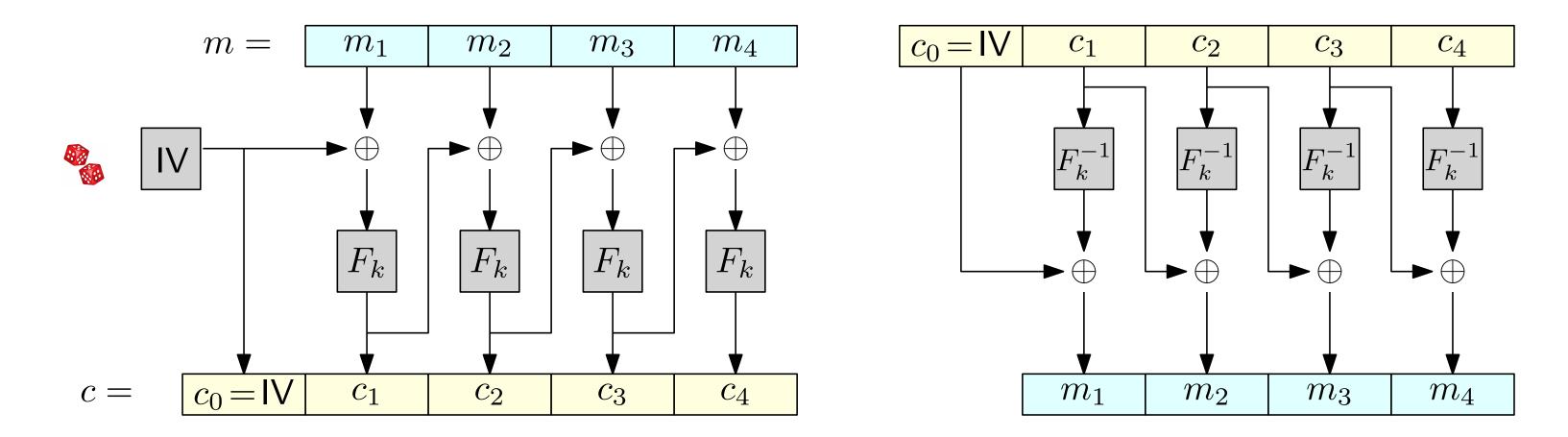
- To decrypt m_i we need c_{i-1}
- $m_i = F_k^{-1}(c_i) \oplus c_{i-1}$



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



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)

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.

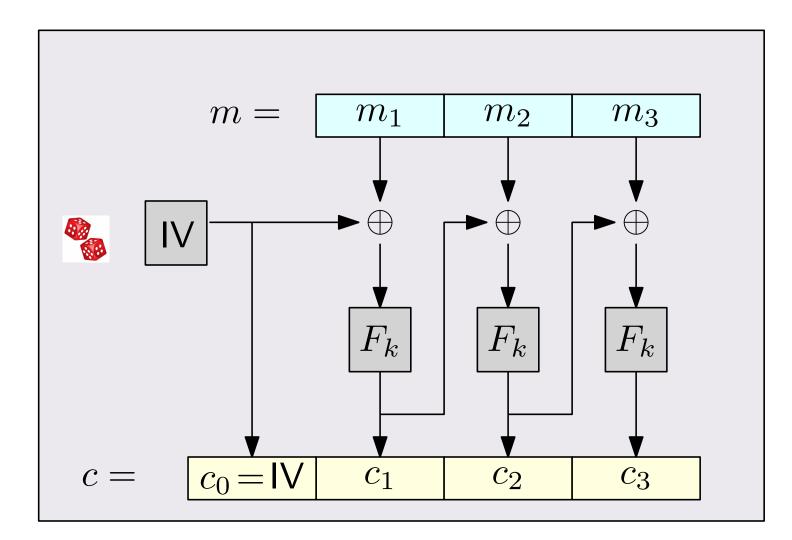


*But, depending on the implementation, it might be vulnerable to some subtle attacks (not really a fault of the encryption scheme, but something to be aware of)

Chained CBC mode

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

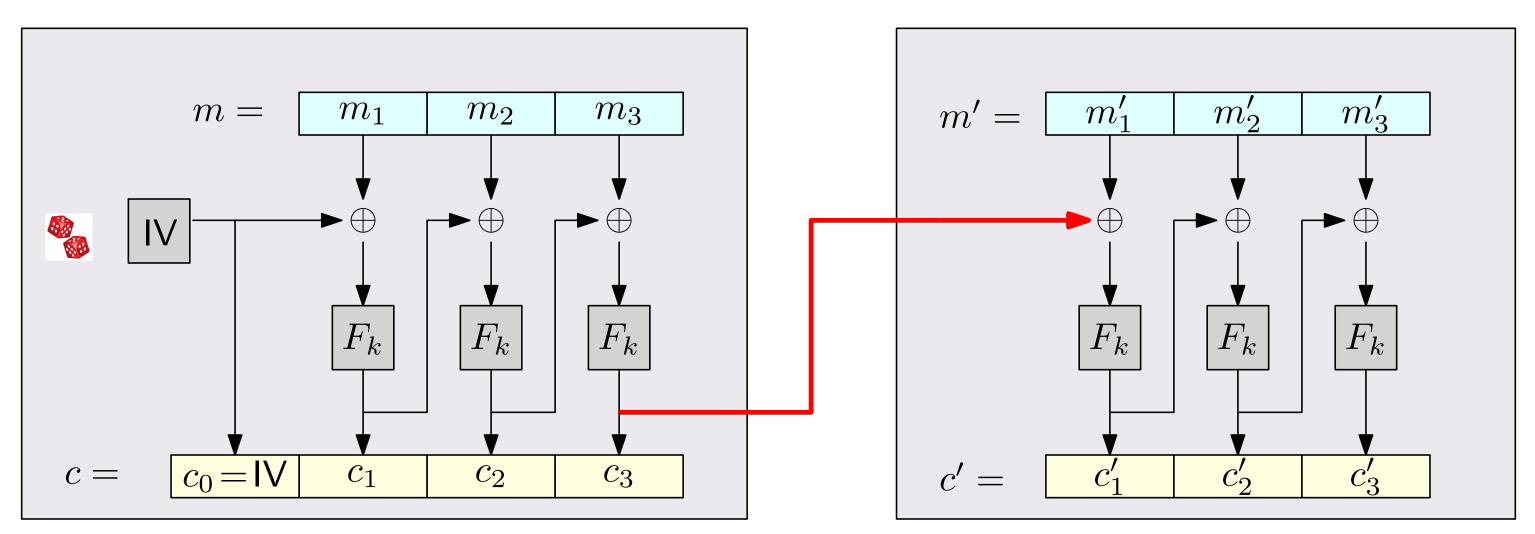
• When the first message is encrypted a random IV is chosen (like in CBC mode)

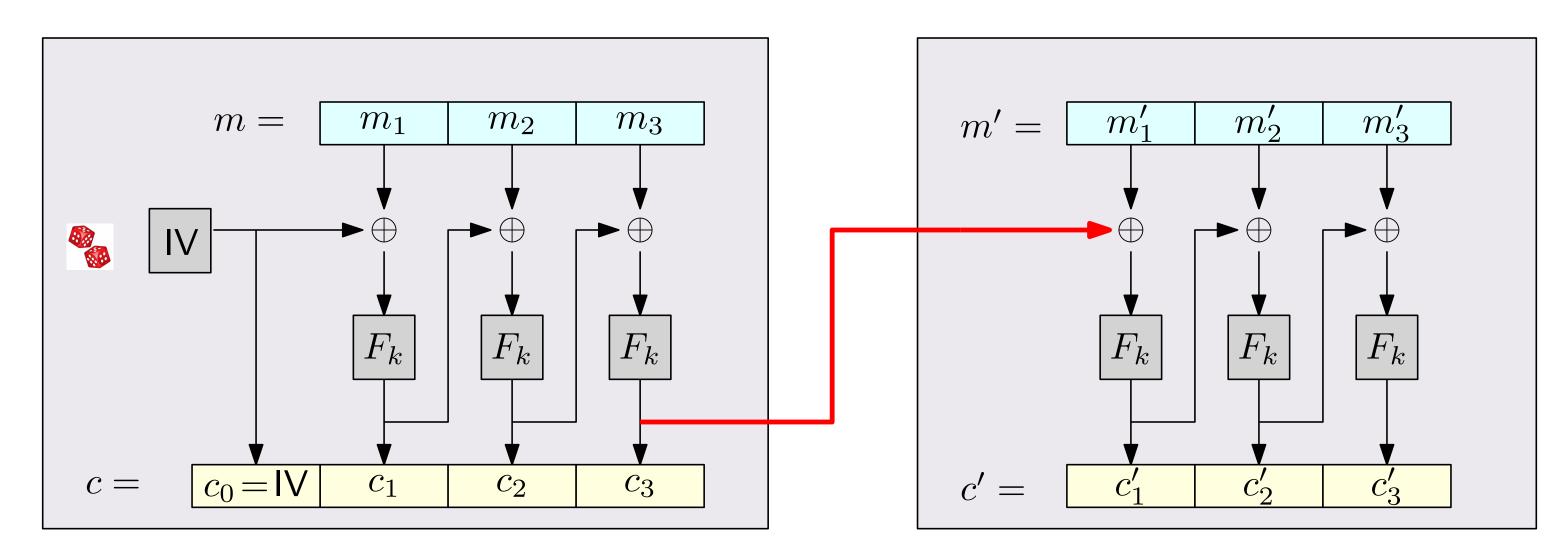


Chained CBC mode

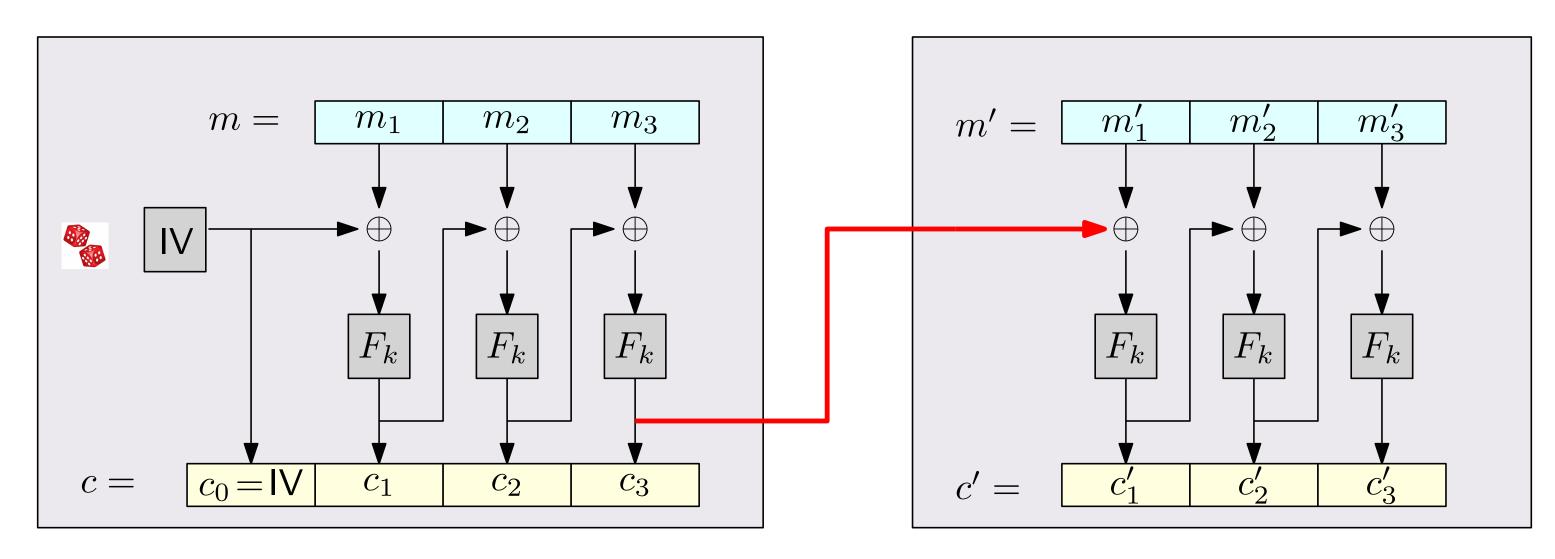
There is a stateful variant of CBC called **chained CBC** that handles multiple messages as follows:

- 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



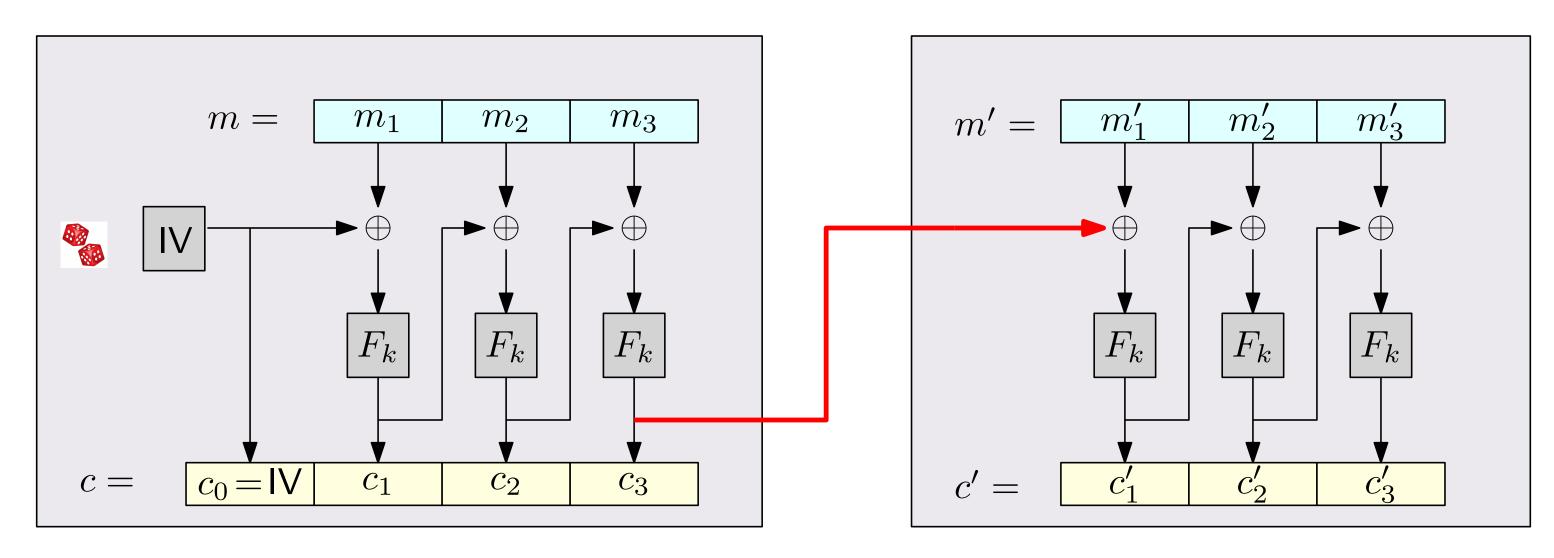


Is chained CBC mode CPA-secure?



Is chained CBC mode CPA-secure?

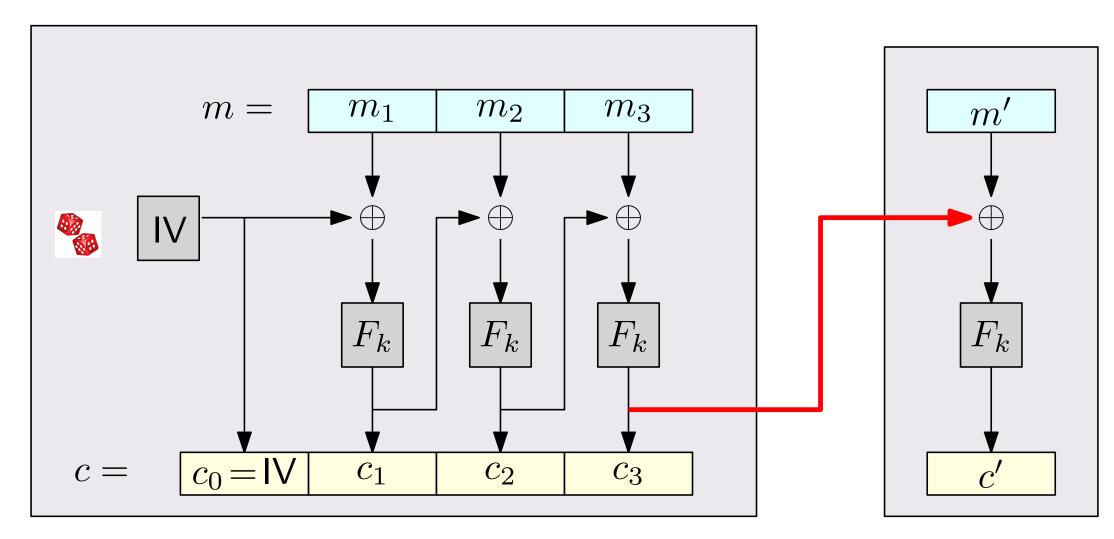
We are just simulating CBC mode on a bigger message $m || m' \dots$



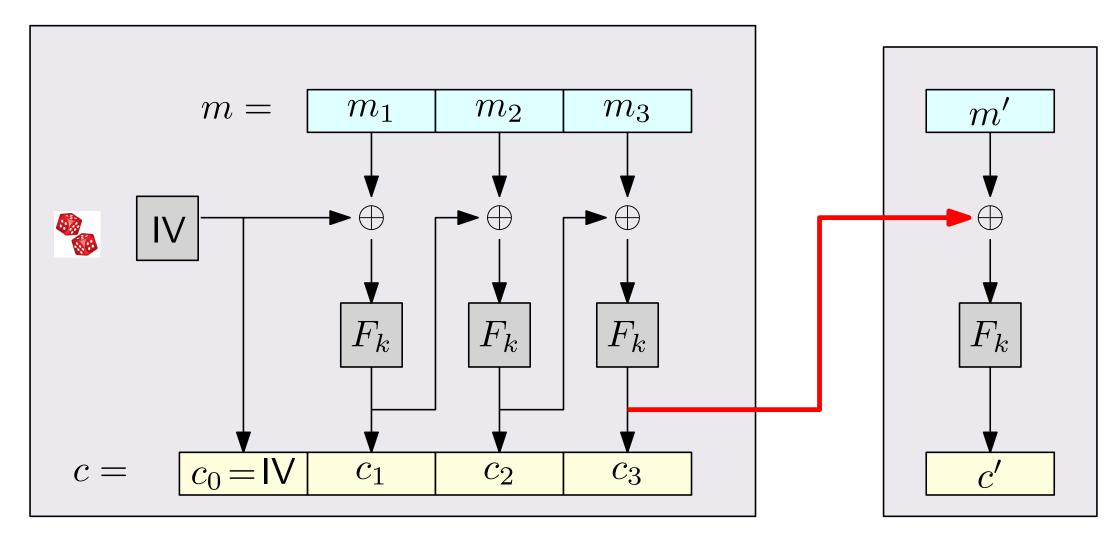
Is chained CBC mode CPA-secure?

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

No!

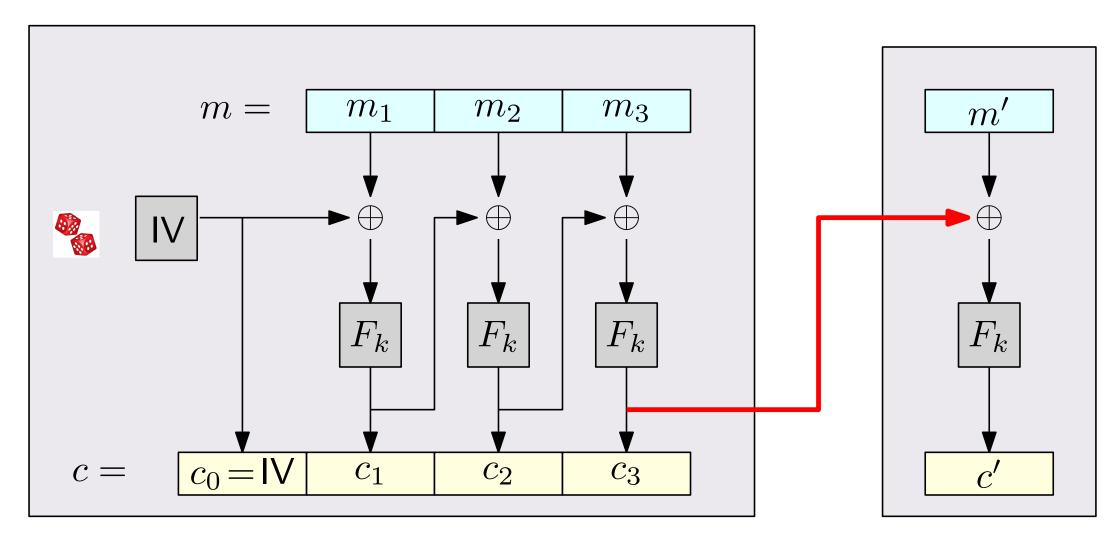


Suppose that the adversary observes c and knows that m_1 is either x or y (e.g., x = ATTACK! and y = RETREAT)



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

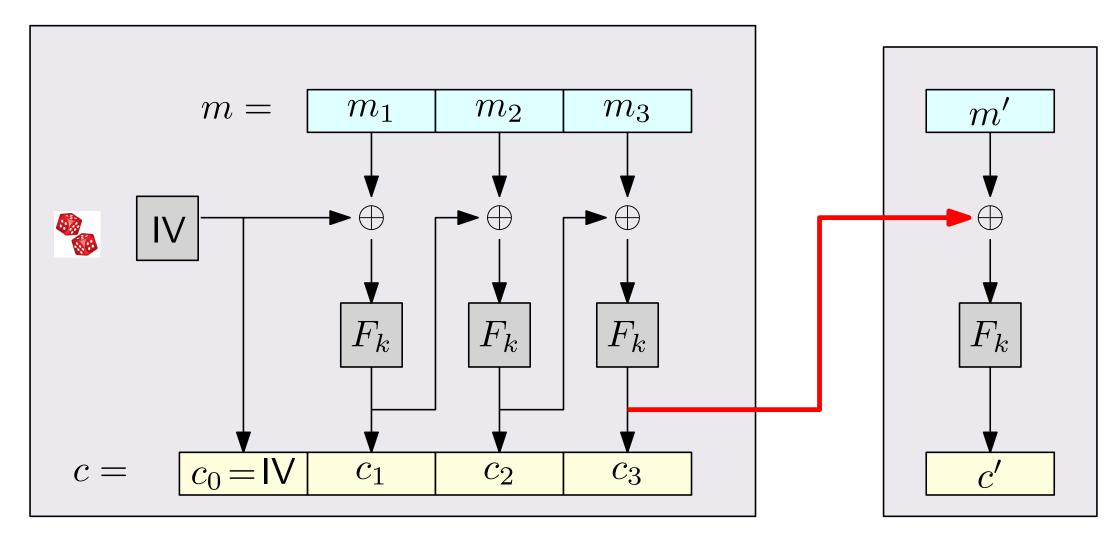
The adversary convinces Alice to encrypt $m' = c_0 \oplus x \oplus c_3$



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 $m' = c_0 \oplus x \oplus c_3$

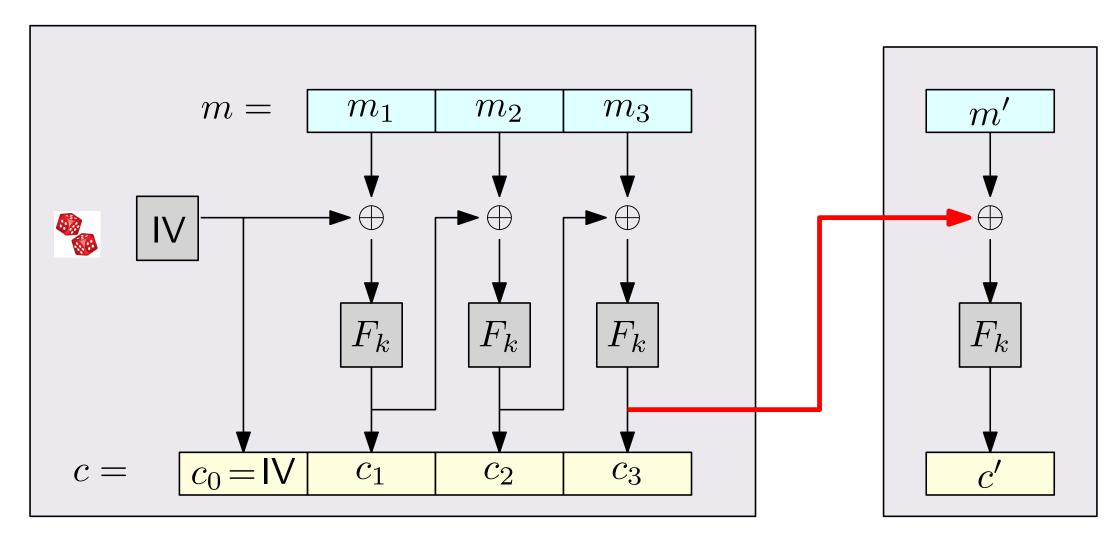
If $m_1 = x$ then $c' = F_k(c_3 \oplus m')$



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 $m' = c_0 \oplus x \oplus c_3$

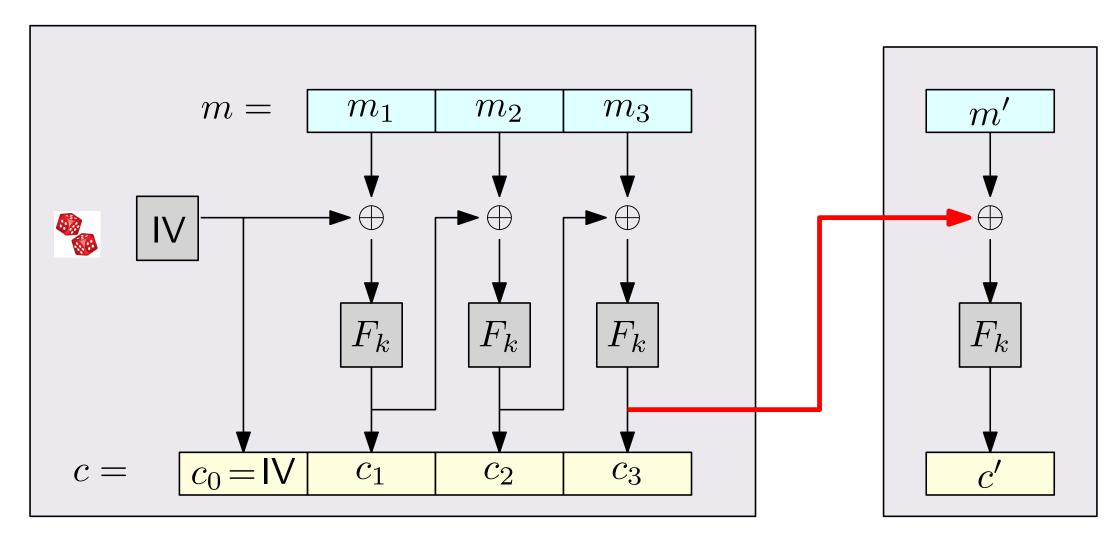
If $m_1 = x$ then $c' = F_k(c_3 \oplus m') = F_k(c_3 \oplus c_0 \oplus x \oplus c_3)$



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 $m' = c_0 \oplus x \oplus c_3$

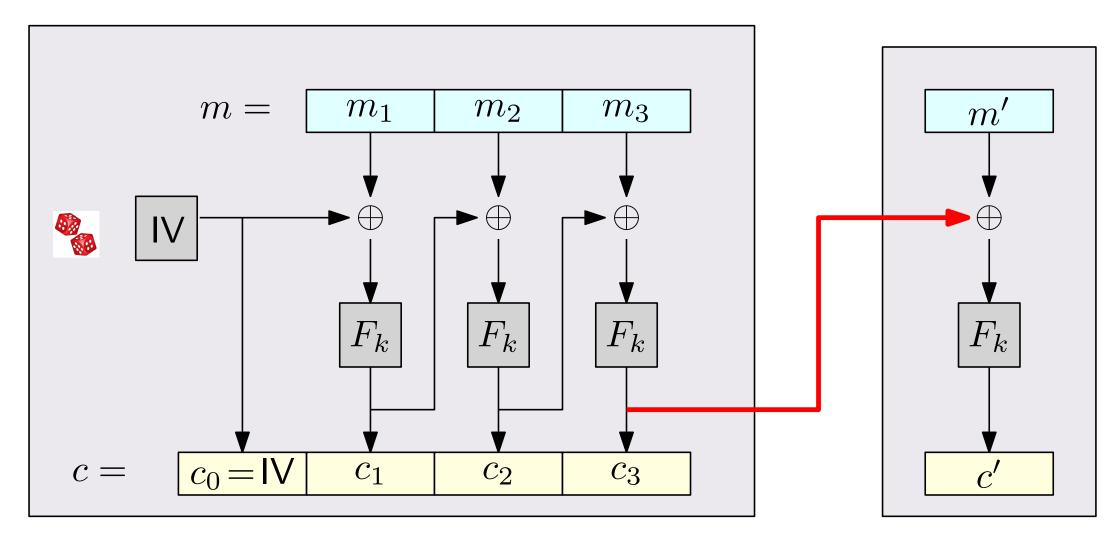
If $m_1 = x$ then $c' = F_k(c_3 \oplus m') = F_k(c_3 \oplus c_0 \oplus x \oplus c_3) = F_k(c_0 \oplus x)$



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

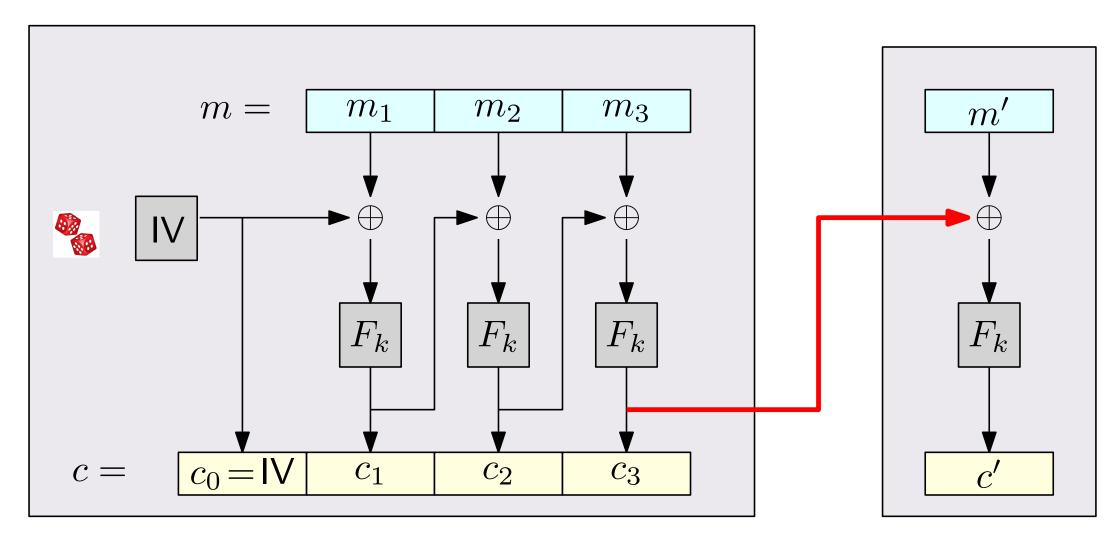


Suppose that the adversary observes c and knows that m_1 is either x or y (e.g., x = ATTACK! and y = RETREAT)

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

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

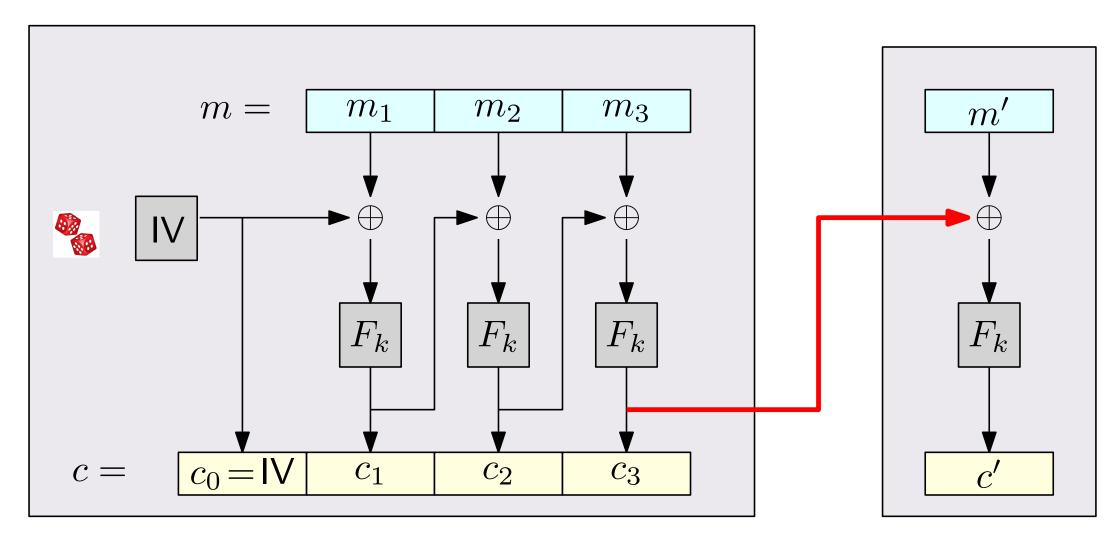
If $m_1 \neq x$ then $c' = F_k(c_3 \oplus m')$



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

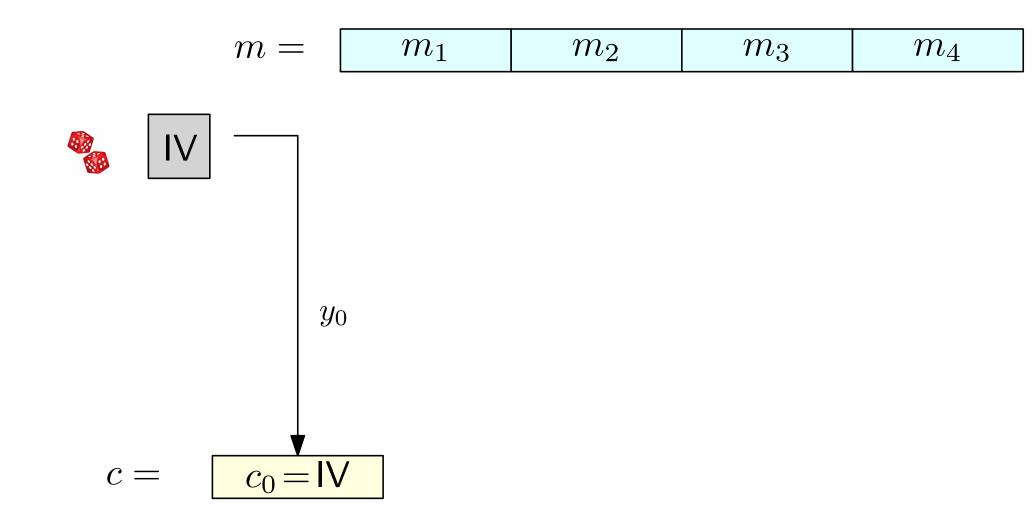


Suppose that the adversary observes c and knows that m_1 is either x or y (e.g., x = ATTACK! and y = RETREAT)

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

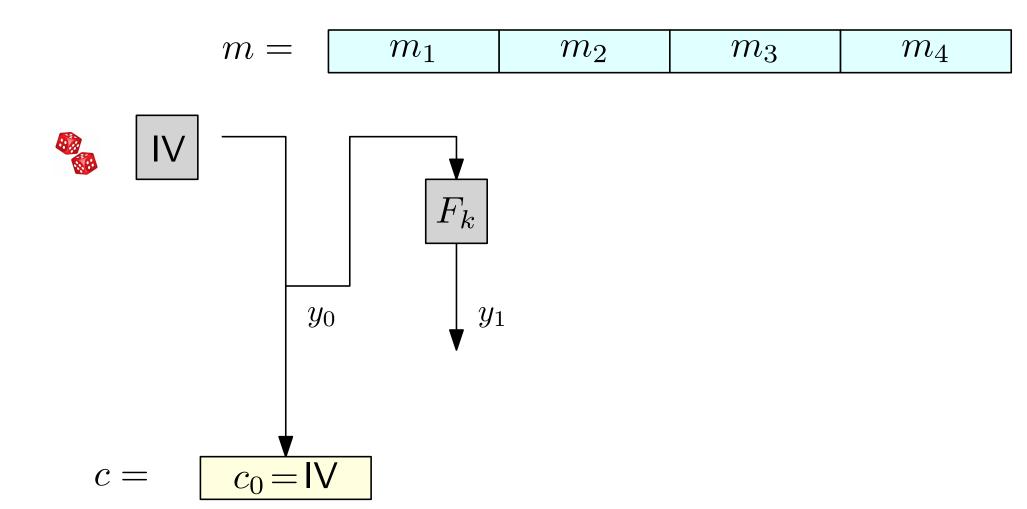
If $m_1 = x$ then $c' = F_k(c_3 \oplus m') = F_k(c_3 \oplus c_0 \oplus x \oplus c_3) = F_k(c_0 \oplus x) = F_k(c_0 \oplus m_1) = c_1$

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

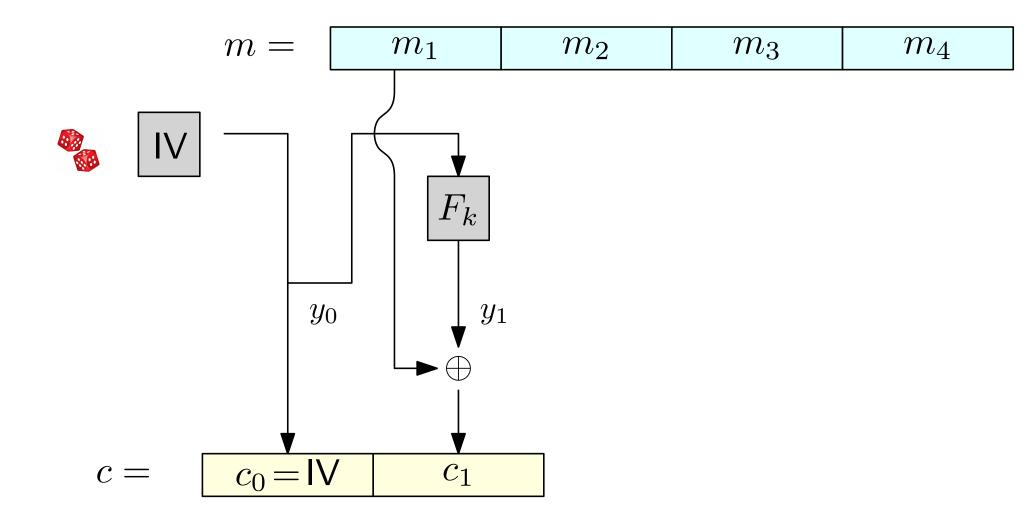


Encrypting:

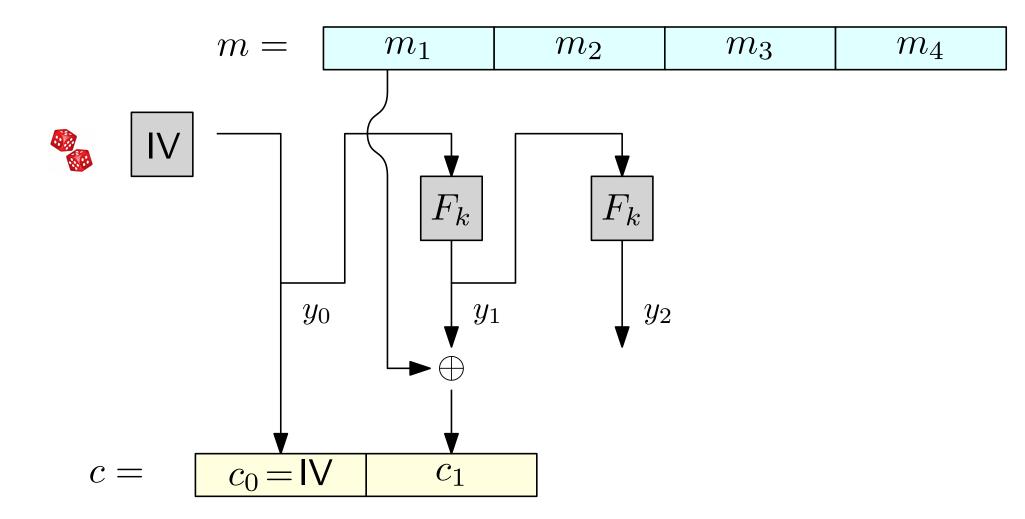
• A random IV is chosen and sent as the first block c_0 of the ciphertext. Let $y_0 = c_0 = IV$



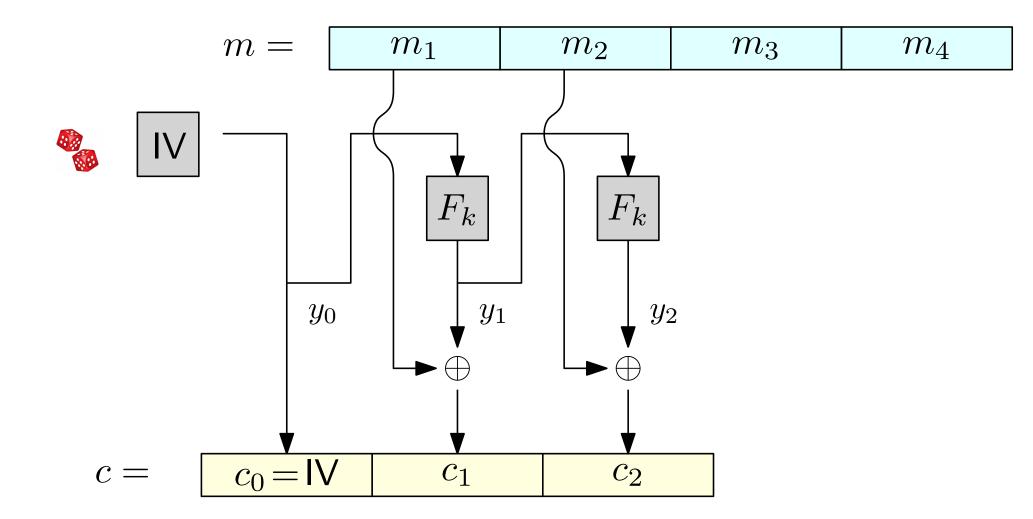
- A random IV is chosen and sent as the first block c_0 of the ciphertext. Let $y_0 = c_0 = IV$
- $y_i = F_k(y_{i-1})$



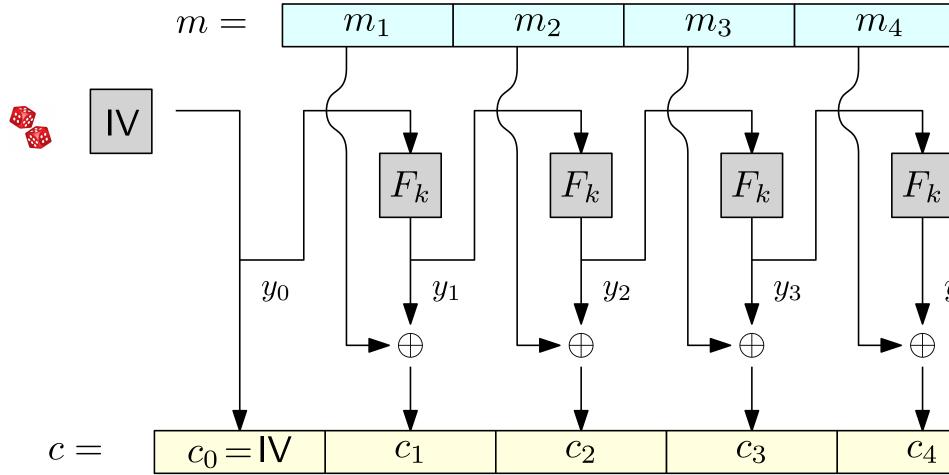
- A random IV is chosen and sent as the first block c_0 of the ciphertext. Let $y_0 = c_0 = IV$
- $y_i = F_k(y_{i-1})$
- $c_i = y_i \oplus m_i$



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



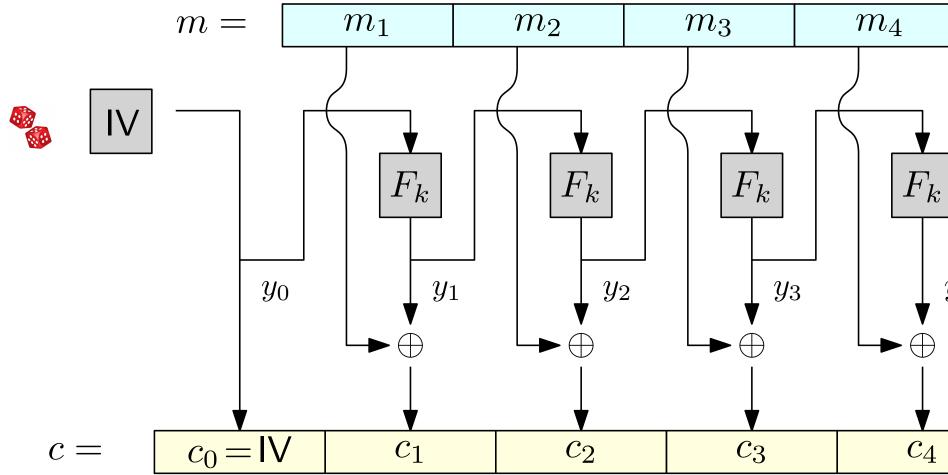
- A random IV is chosen and sent as the first block c_0 of the ciphertext. Let $y_0 = c_0 = IV$
- $y_i = F_k(y_{i-1})$
- $c_i = y_i \oplus m_i$



Encrypting:

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

 y_4

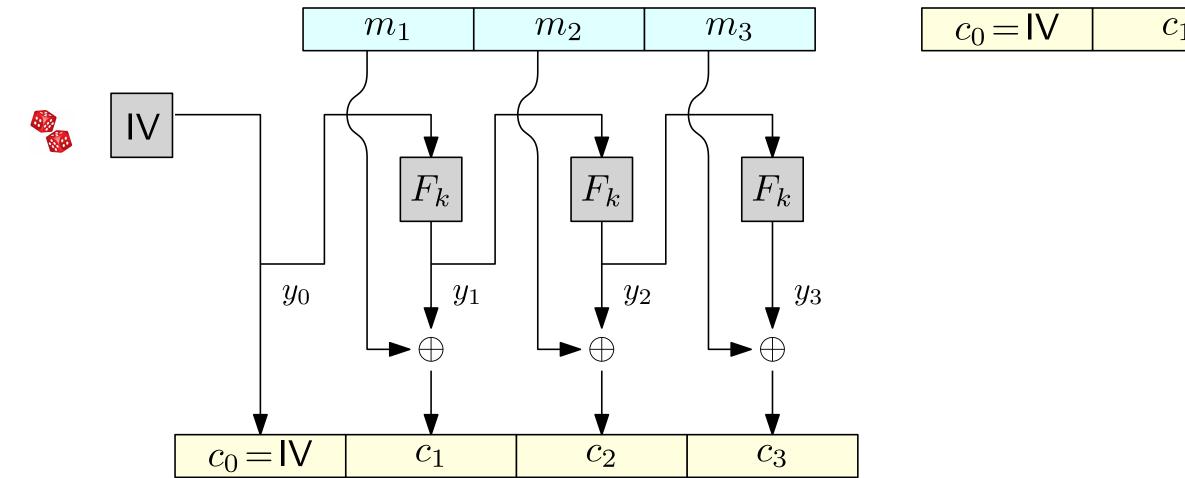


Encrypting:

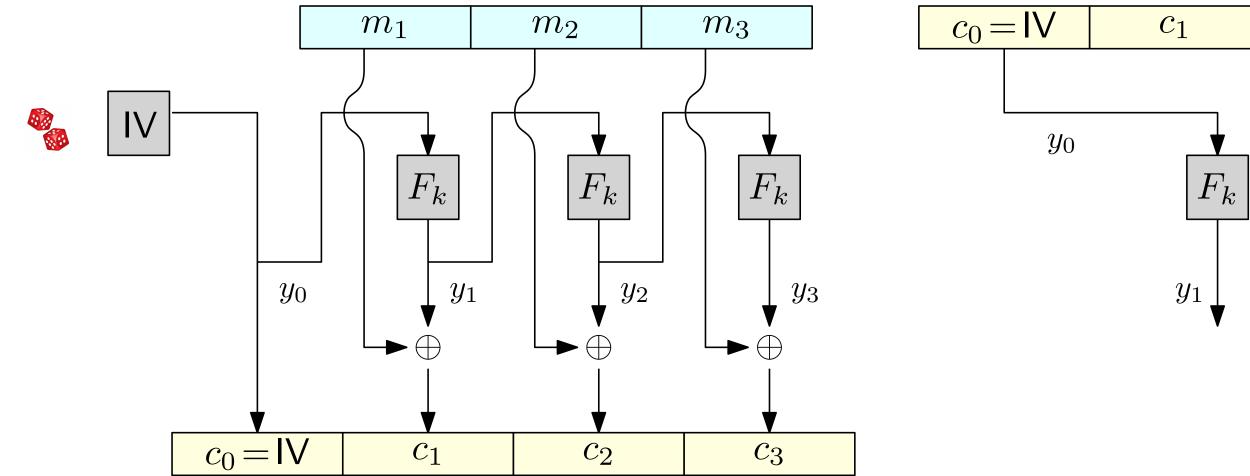
- A random IV is chosen and sent as the first block c_0 of the ciphertext. Let $y_0 = c_0 = IV$
- $y_i = F_k(y_{i-1})$
- $c_i = y_i \oplus m_i$

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

 y_4

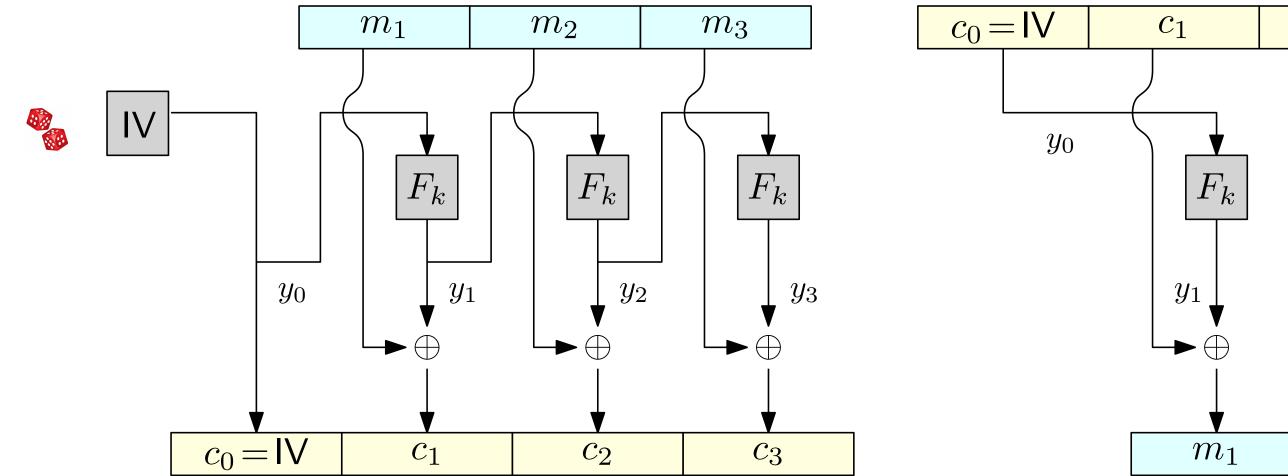


1	c_2	c_3
---	-------	-------



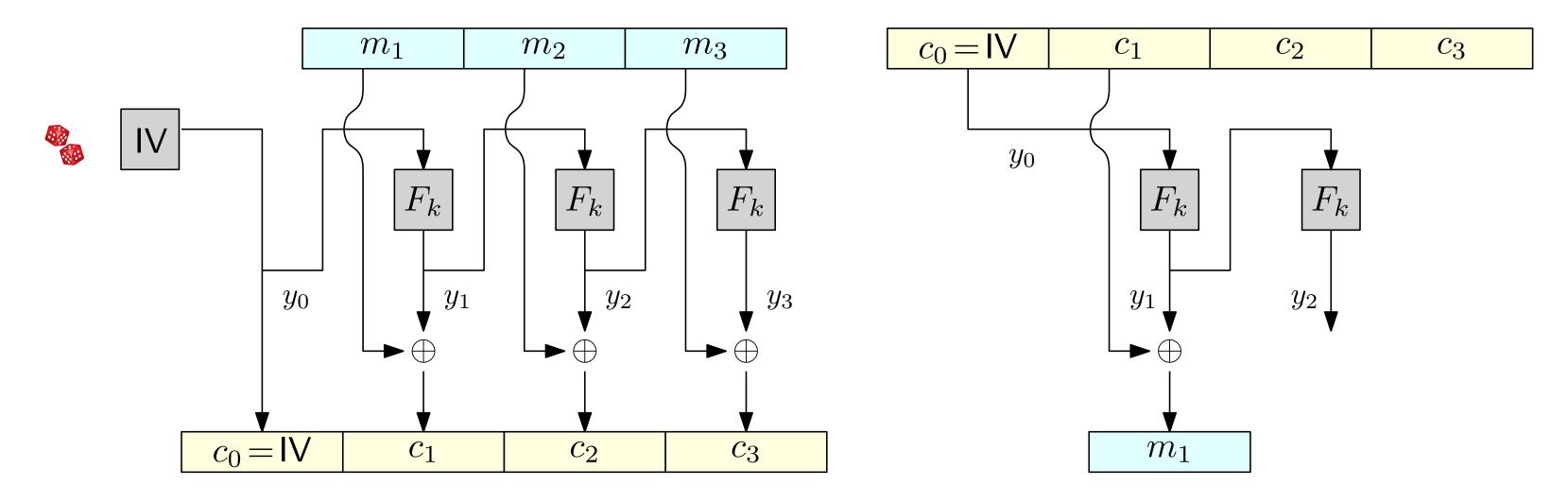
- $y_0 = c_0$
- $y_i = F_k(y_{i-1})$

1 c_2	c_3
-----------	-------

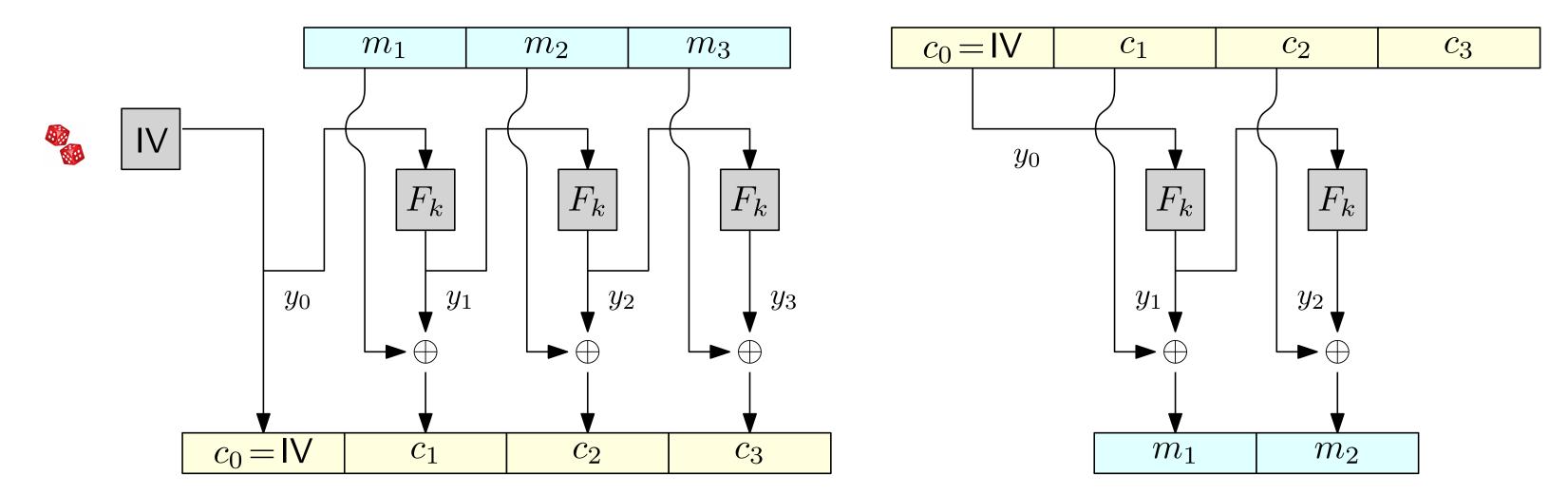


- $y_0 = c_0$
- $y_i = F_k(y_{i-1})$
- $m_i = y_i \oplus c_i$

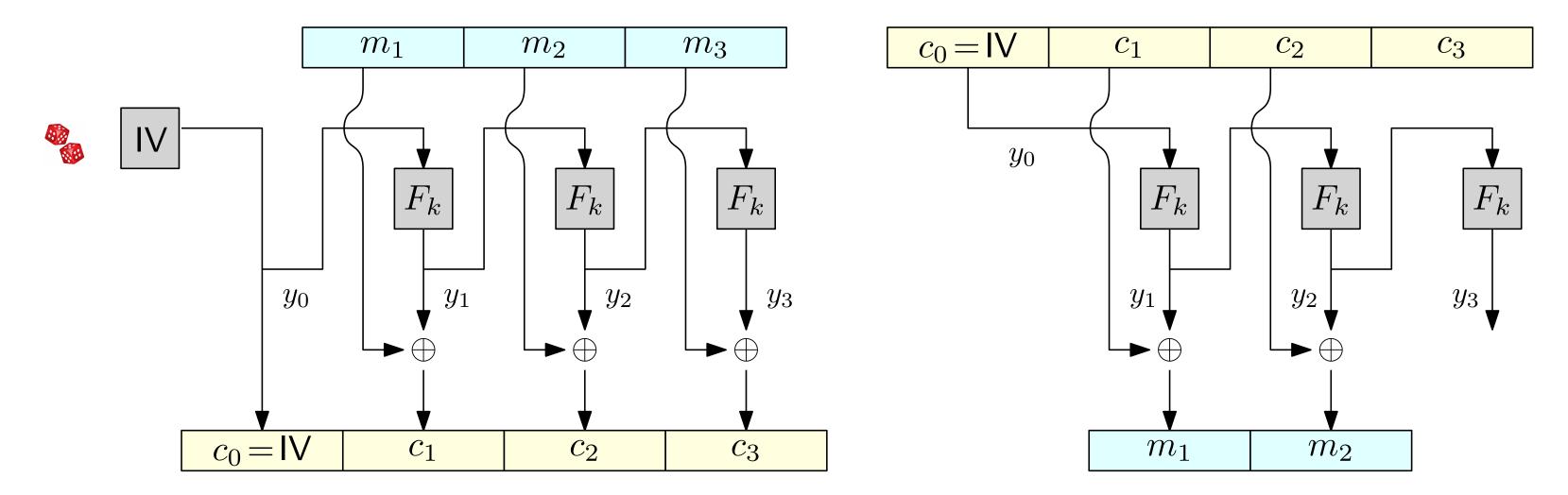
$1 c_2 c_3$



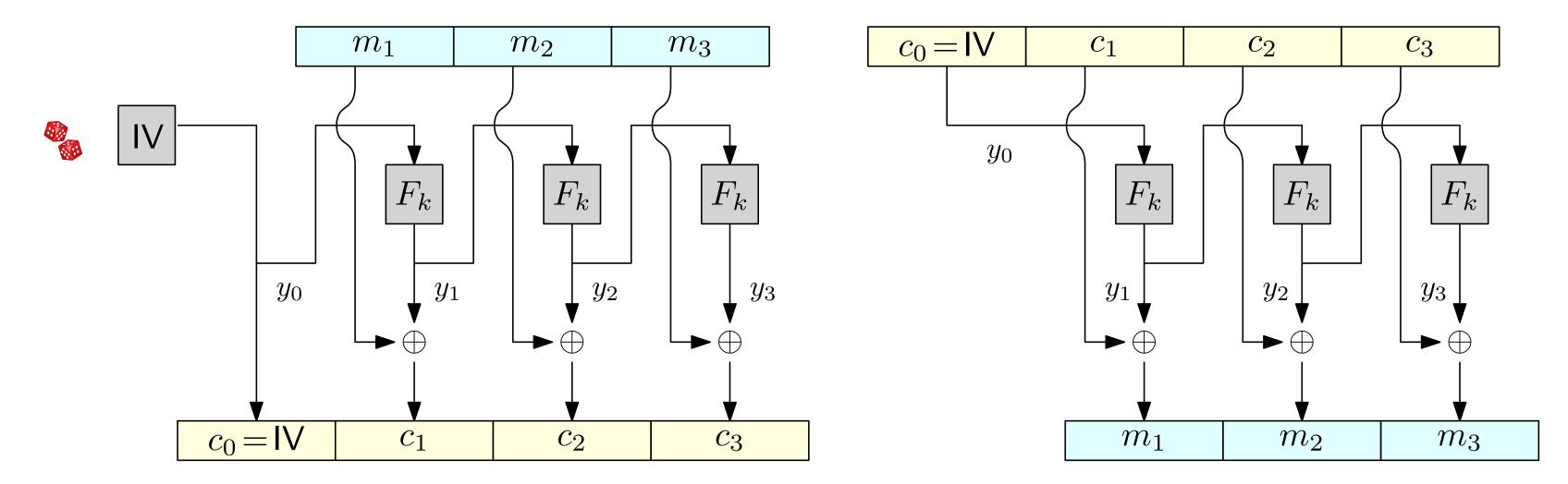
- $y_0 = c_0$
- $y_i = F_k(y_{i-1})$
- $m_i = y_i \oplus c_i$



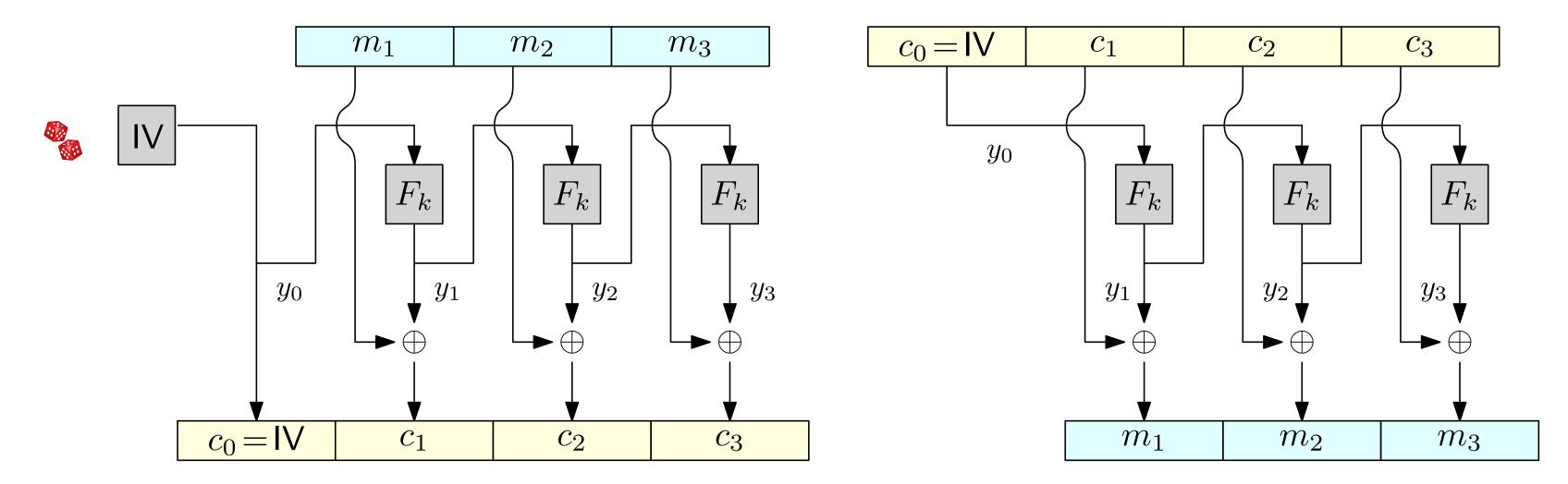
- $y_0 = c_0$
- $y_i = F_k(y_{i-1})$
- $m_i = y_i \oplus c_i$



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



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



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

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

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

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})

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).

Is OFB mode CPA-secure?

(notice that we never used F^{-1})

Output Feedback (OFB) mode

Encryption and decryption must be done sequentially

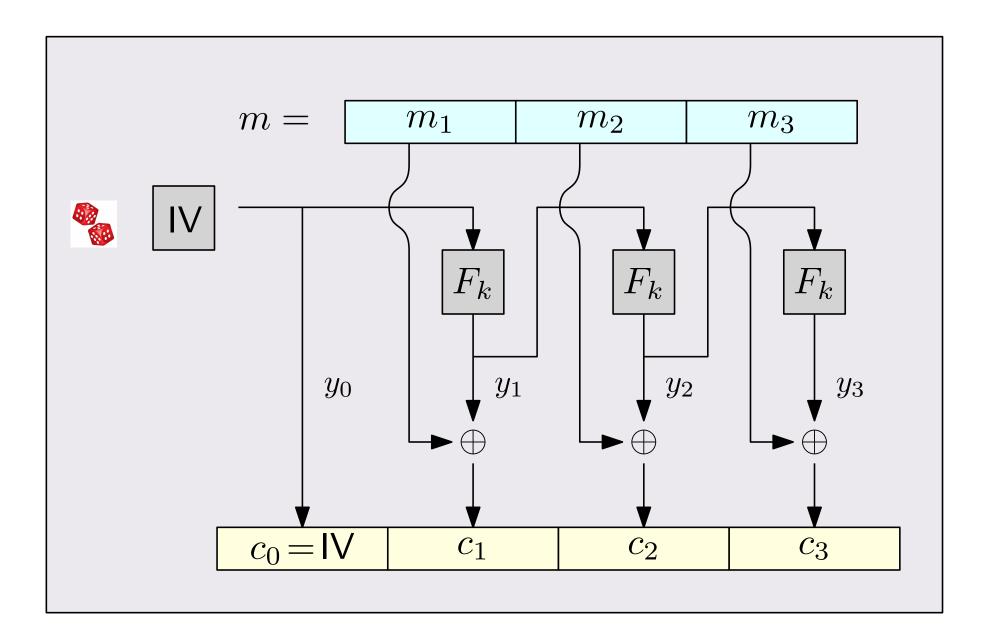
- An optimization: the stream y_1, y_2, y_3, \ldots only depends on the IV (and the key): it can be pre-computed before the message needs to be encrypted
- If the last block is not full, the ciphertext can be truncated to the plaintext length
- F can be any PRF (not necessarily a PRP).

Is OFB mode CPA-secure?

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

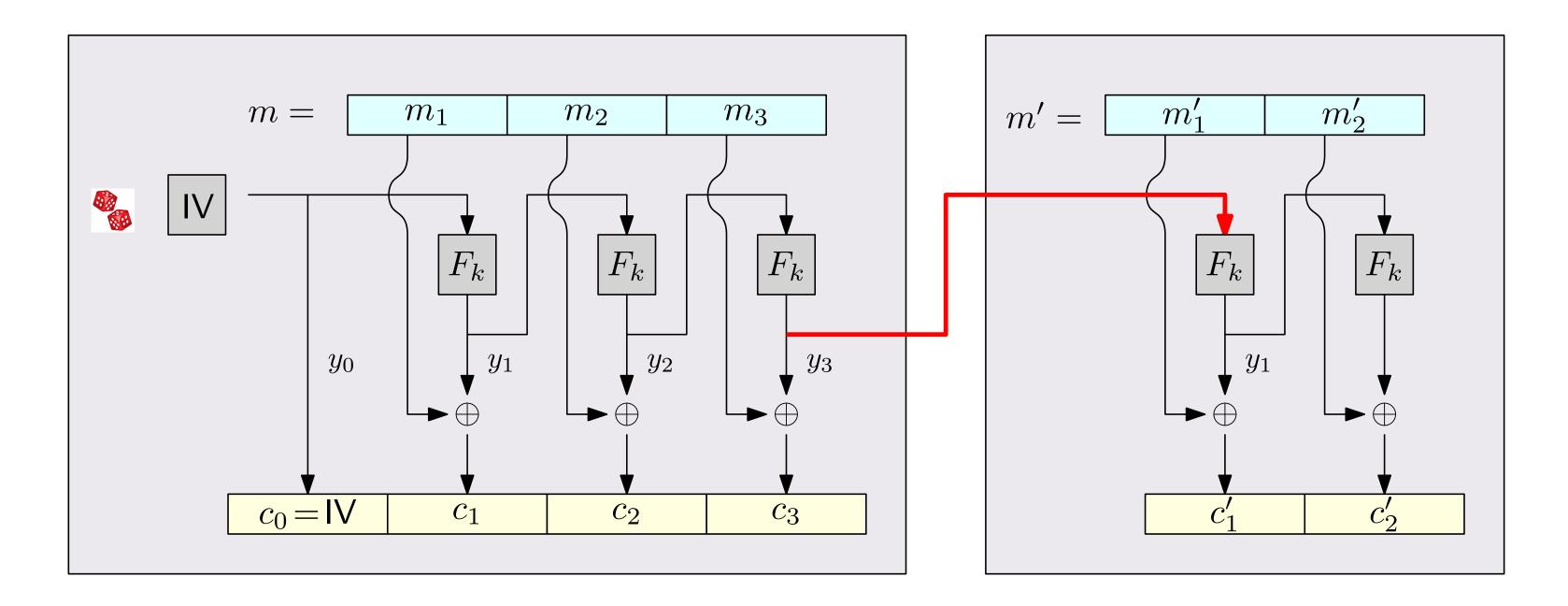
Output Feedback (OFB) mode, stateful variant

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**



Can be viewed as a stream cipher

$$m = \begin{bmatrix} m_1 & m_2 \end{bmatrix}$$

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

|--|

Can be viewed as a stream cipher

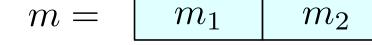
$$m = \begin{bmatrix} m_1 & m_2 \end{bmatrix}$$

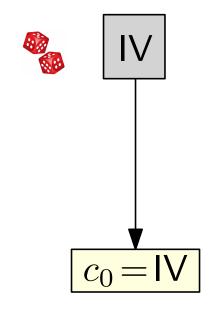
- Split the input to F into an IV and a counter For example:
 - $\mathsf{IV} \in \{0,1\}^{3n/4}$
 - counter $\in \{0,1\}^{n/4}$

|--|

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





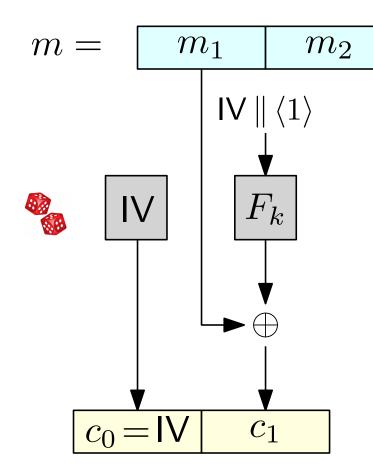
Encrypting:

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

|--|

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



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

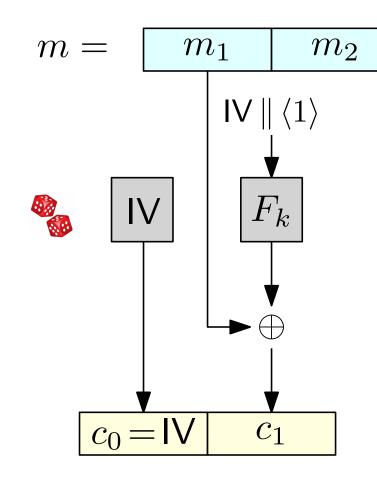
|--|

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

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



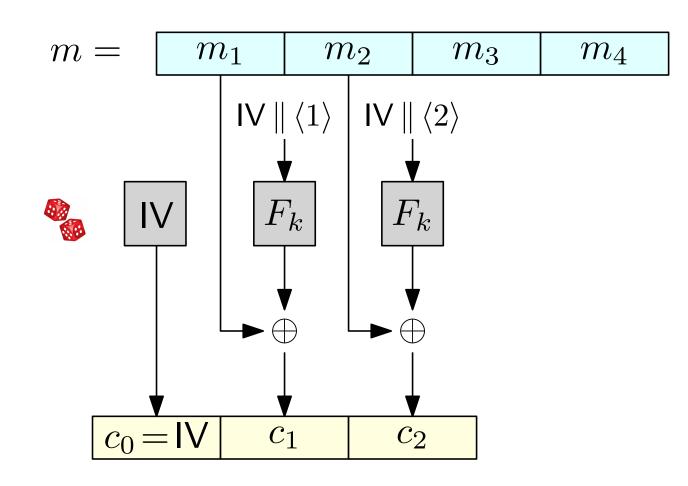
|--|

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

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

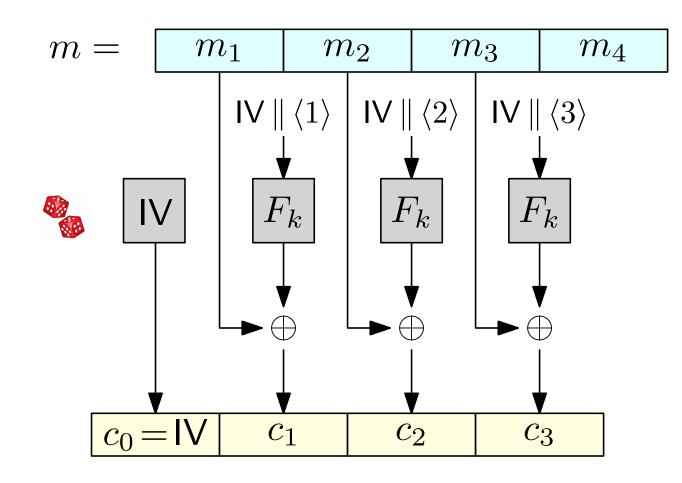


Can be viewed as a stream cipher

- 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

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

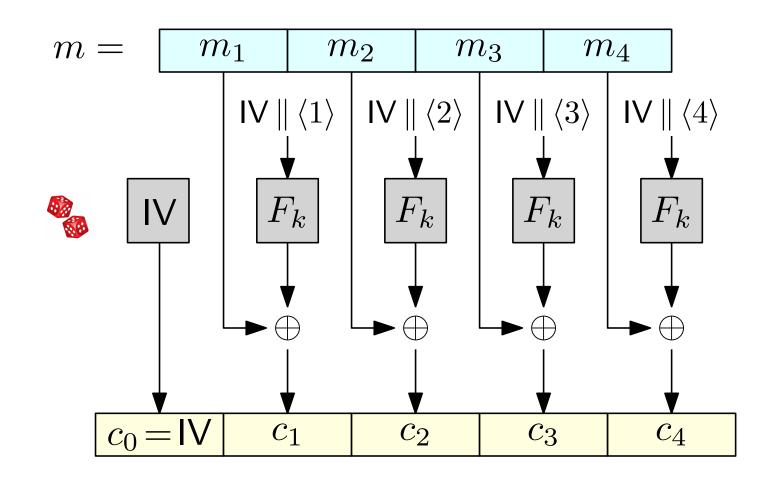


Can be viewed as a stream cipher

- 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

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



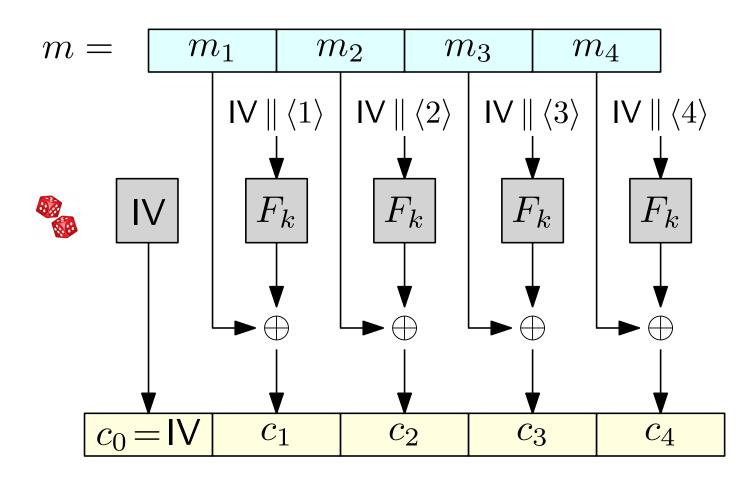
Can be viewed as a stream cipher

- 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 ciphertext.

• Set the IV to the first block c_0 of the

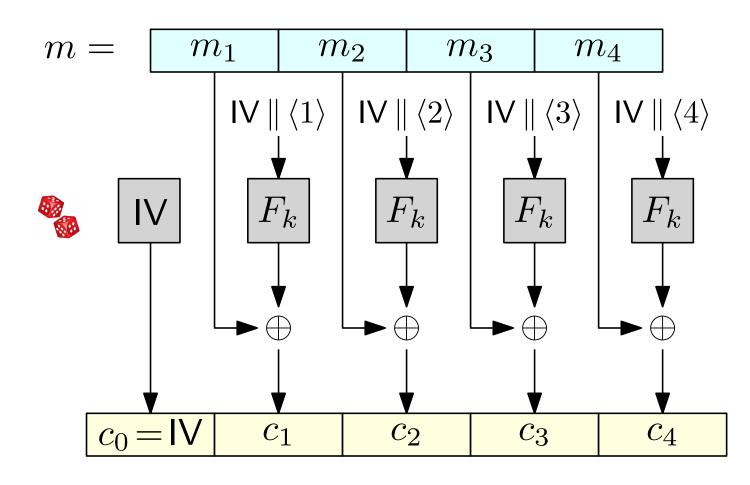
Can be viewed as a stream cipher

- 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 ciphertext.

•
$$m_i = F_k(\mathsf{IV} \mid$$

• Set the IV to the first block c_0 of the

 $\langle i
angle) \oplus c_i$

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

- 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!

- 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)

- 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)

- 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)

Is CTR mode CPA-secure?

- 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)

Is CTR mode CPA-secure?

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

- 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)

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

 $\mathsf{IV} = 00 \dots 000, \ 00 \dots 001, \ 00 \dots 010, \ 00 \dots 011, \ \dots$