Random Numbers

Our goal is to turn a plaintext into a ciphertext that looks *random*

Random Numbers

Our goal is to turn a plaintext into a ciphertext that looks *random*

... but isn't really random

Random Numbers

Our goal is to turn a plaintext into a ciphertext that looks *random*

... but isn't really random

Doesn't that sound like a pseudo-random number generator (PRNG)?

PRNGs

```
> random(256)
243
> random(256)
122
> random(256)
187
> random(256)
43
> random(256)
200
> random(256)
229
```

PRNGs

```
> random.seed(74658319934)
> random(256)
243
> random(256)
122
> random(256)
187
> random(256)
43
> random(256)
200
> random(256)
229
```

Stream Cipher

Let $K = K_0, K_1, \ldots K_i, \ldots$ be a sequence of random numbers that is generated from a seed 

Stream Cipher

Let $K = K_0, K_1, \ldots K_i, \ldots$ be a sequence of random numbers that is generated from a seed 

The ith ciphertext CT element from the ith plaintext PT element:

$$CT_i = PT_i \oplus K_i$$

xor

Stream Cipher

Let $K = K_0, K_1, \ldots K_i, \ldots$ be a sequence of random numbers that is generated from a seed 

The ith ciphertext CT element from the ith plaintext PT element:

$$CT_i = PT_i \oplus K_i$$

Stream Cipher

Let $K = K_0, K_1, \ldots K_i, \ldots$ be a sequence of random numbers that is generated from a seed 

The ith ciphertext CT element from the ith plaintext PT element:

$$CT_i = PT_i \oplus K_i$$

Since $\oplus$ is its own inverse:

$$PT_i = CT_i \oplus K_i$$

Stream Cipher

Let $K = K_0, K_1, \ldots K_i, \ldots$ be a sequence of random numbers that is generated from a seed 

The ith ciphertext CT element from the ith plaintext PT element:

$$CT_i = PT_i \oplus K_i$$

Since $\oplus$ is its own inverse:

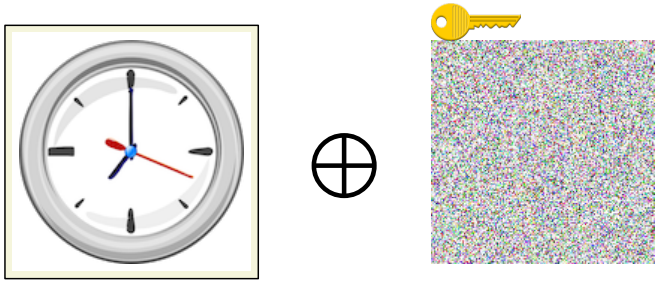
$$PT_i = CT_i \oplus K_i$$

Treating $\oplus$ as mapping over sequences

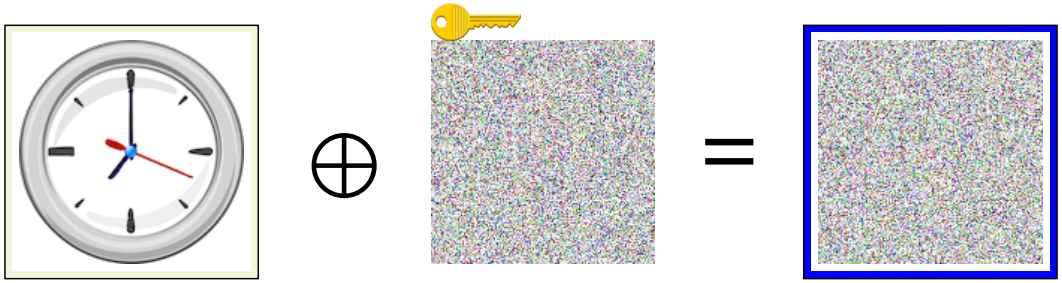
$$CT = PT \oplus K$$

$$PT = CT \oplus K$$

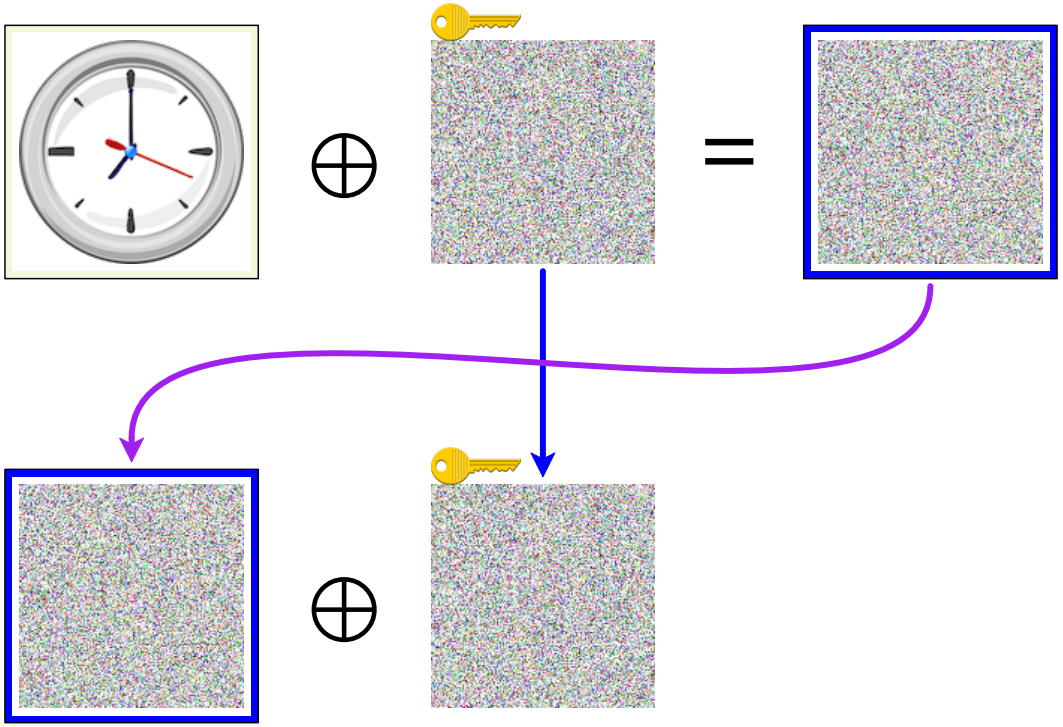
Stream Cipher on an Image



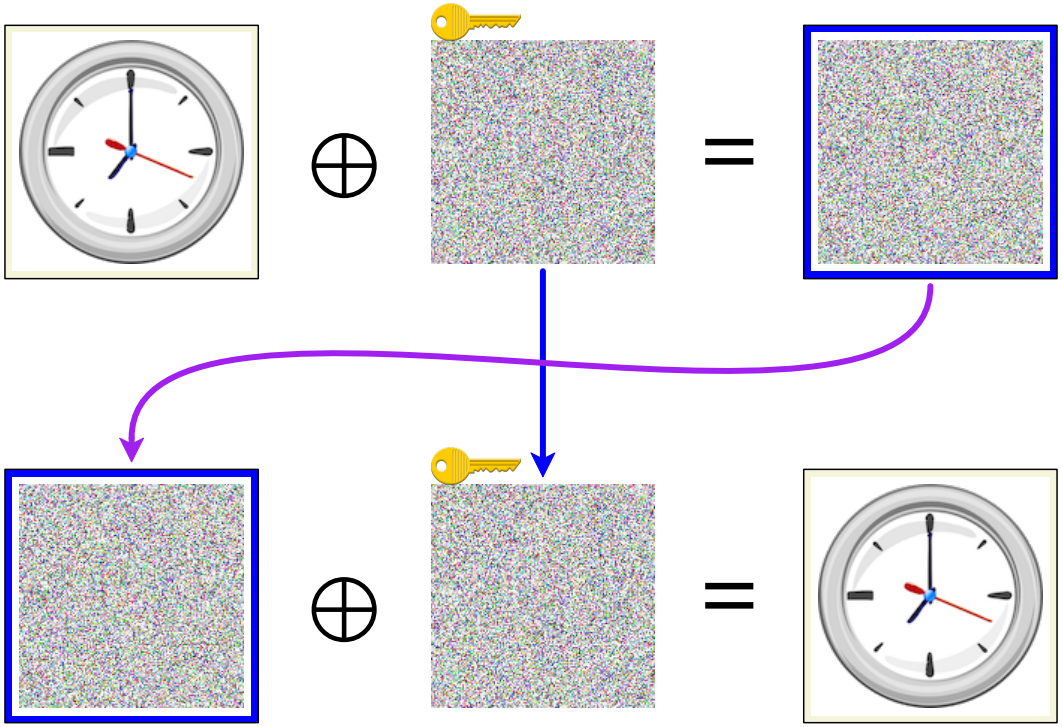
Stream Cipher on an Image



Stream Cipher on an Image



Stream Cipher on an Image



Well, That Was Easy!

Why aren't we done?

Well, That Was Easy!

Why aren't we done?

Problem #1: it turns out that $\oplus$ uses up a key

unless the PRNG has an extra feature making it more general

Well, That Was Easy!

Why aren't we done?

Problem #1: it turns out that $\oplus$ uses up a key

unless the PRNG has an extra feature making it more general

Problem #2: we're getting confidentiality, but not integrity

so, need to combine with something else, not today

Well, That Was Easy!

Why aren't we done?

Problem #1: it turns out that $\oplus$ uses up a key

unless the PRNG has an extra feature making it more general

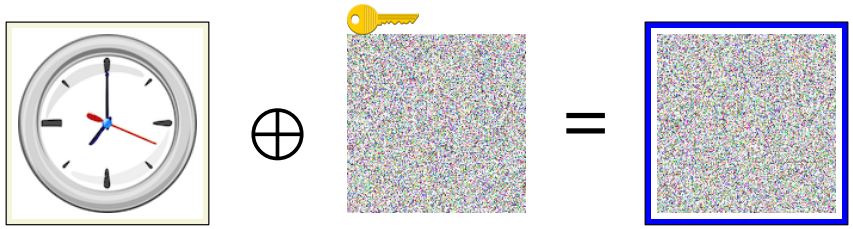
Problem #2: we're getting confidentiality, but not integrity

so, need to combine with something else, not today

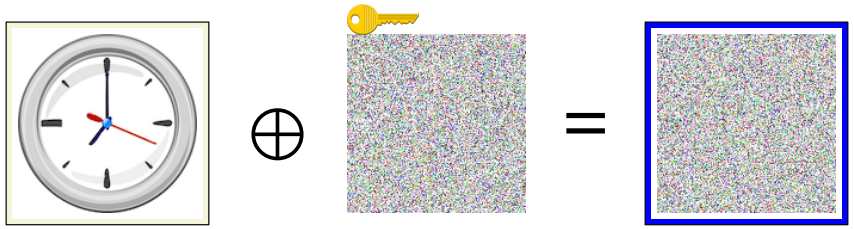
Problem #3: we need a really good PRNG

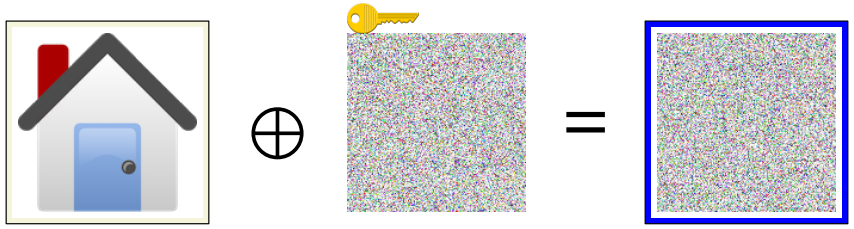
a general building block, so we'll look more today

Misuse of Stream Cipher and Key

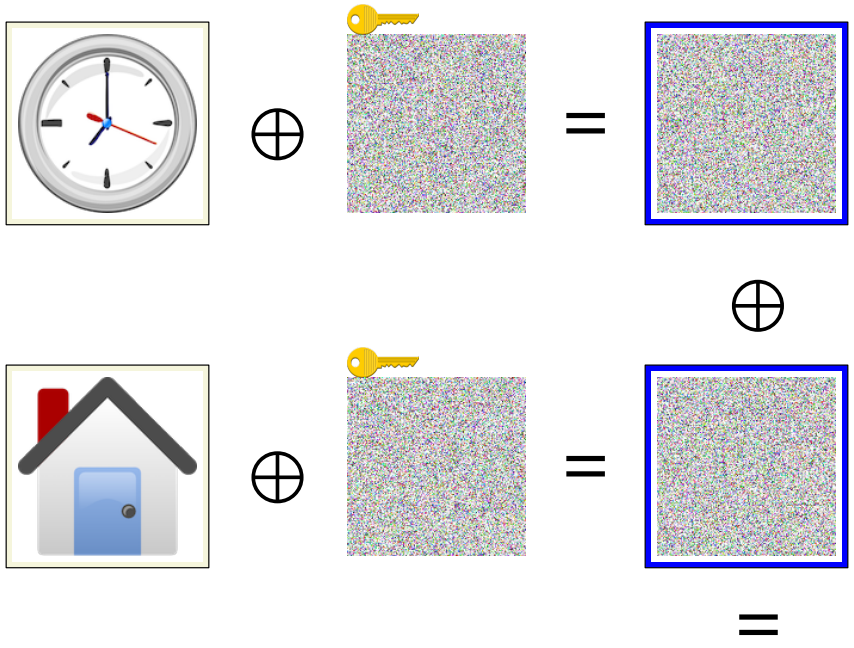


Misuse of Stream Cipher and Key

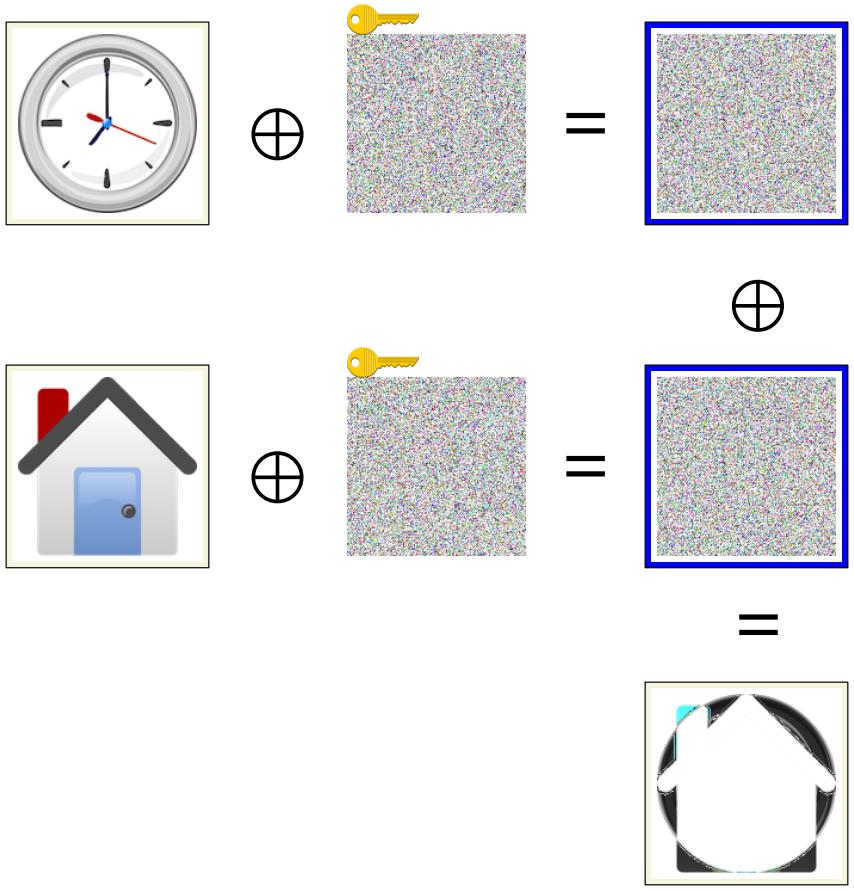




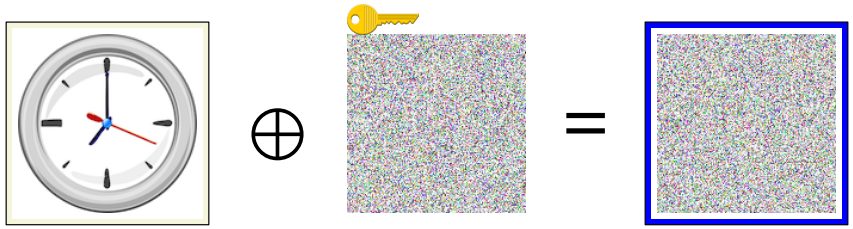
Misuse of Stream Cipher and Key

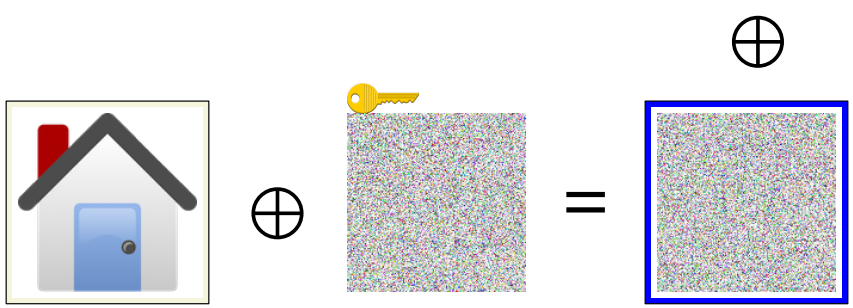


Misuse of Stream Cipher and Key



Misuse of Stream Cipher and Key



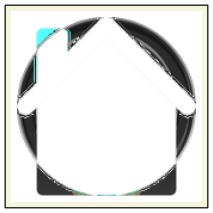


$CT_a = PT_a \oplus K$

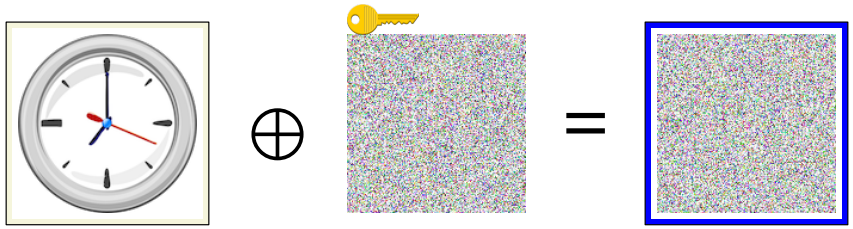
$CT_b = PT_b \oplus K$

$CT_a \oplus CT_b = PT_a \oplus K \oplus PT_b \oplus K$

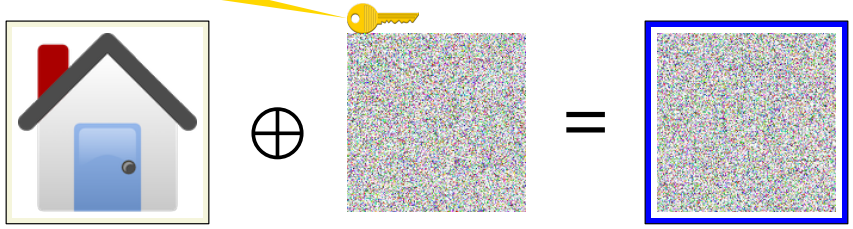
$= PT_a \oplus PT_b$



Misuse of Stream Cipher and Key



Don't use the same key twice!

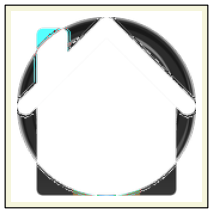


$CT_a = PT_a \oplus K$

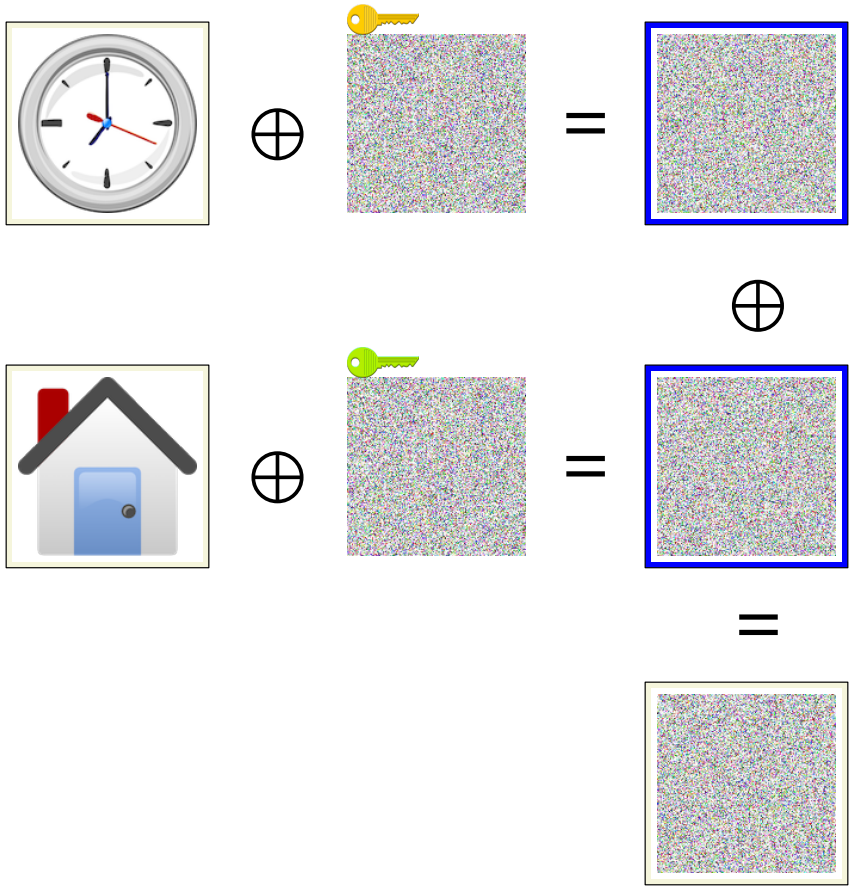
$CT_b = PT_b \oplus K$

$CT_a \oplus CT_b = PT_a \oplus K \oplus PT_b \oplus K$

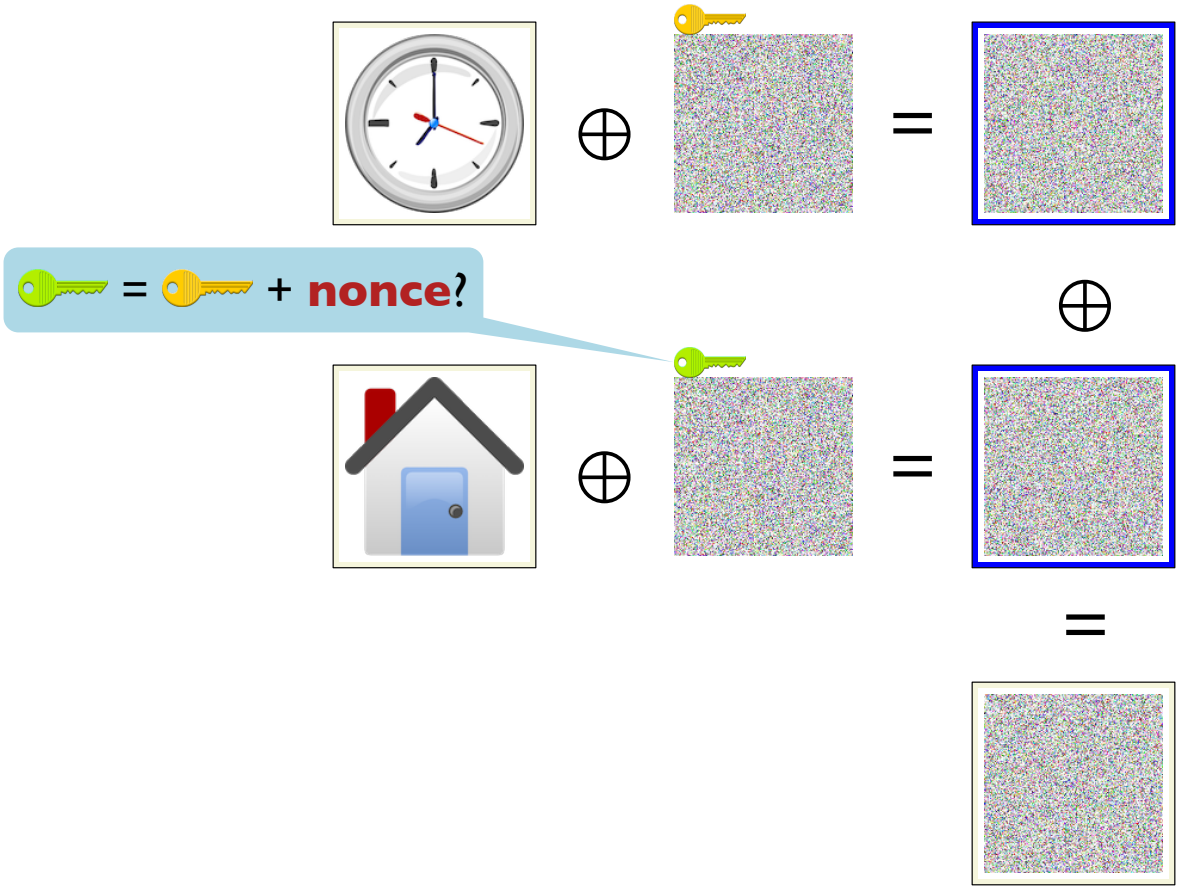
$= PT_a \oplus PT_b$



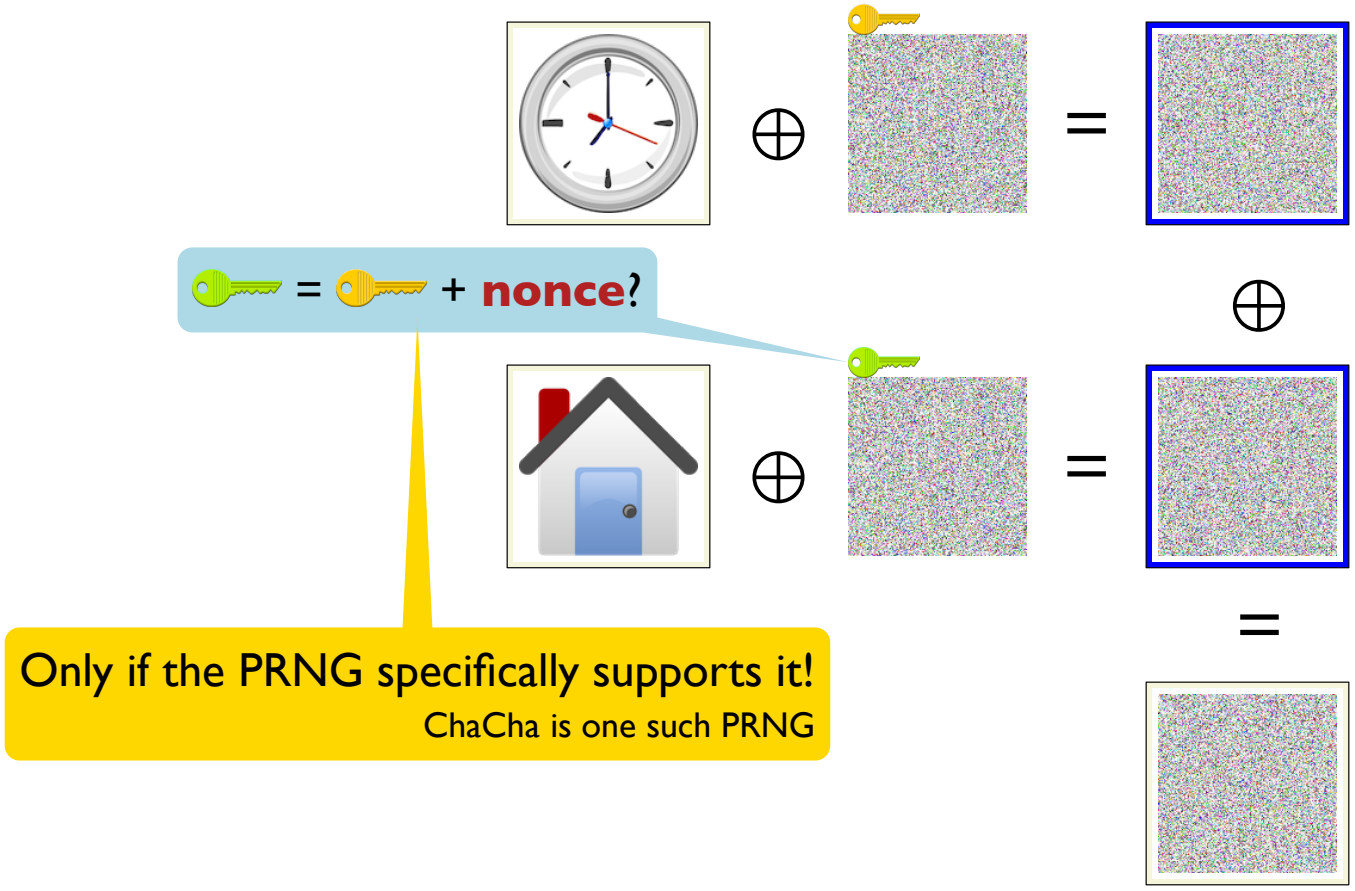
Potentially Correct Use of Stream Cipher and Key



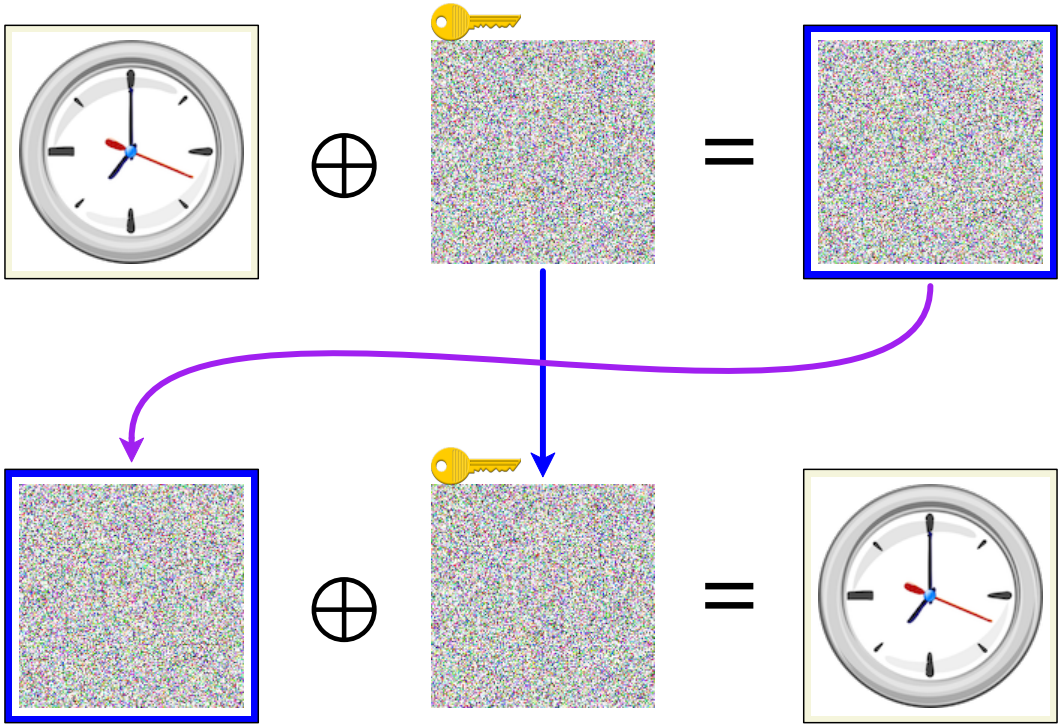
Potentially Correct Use of Stream Cipher and Key



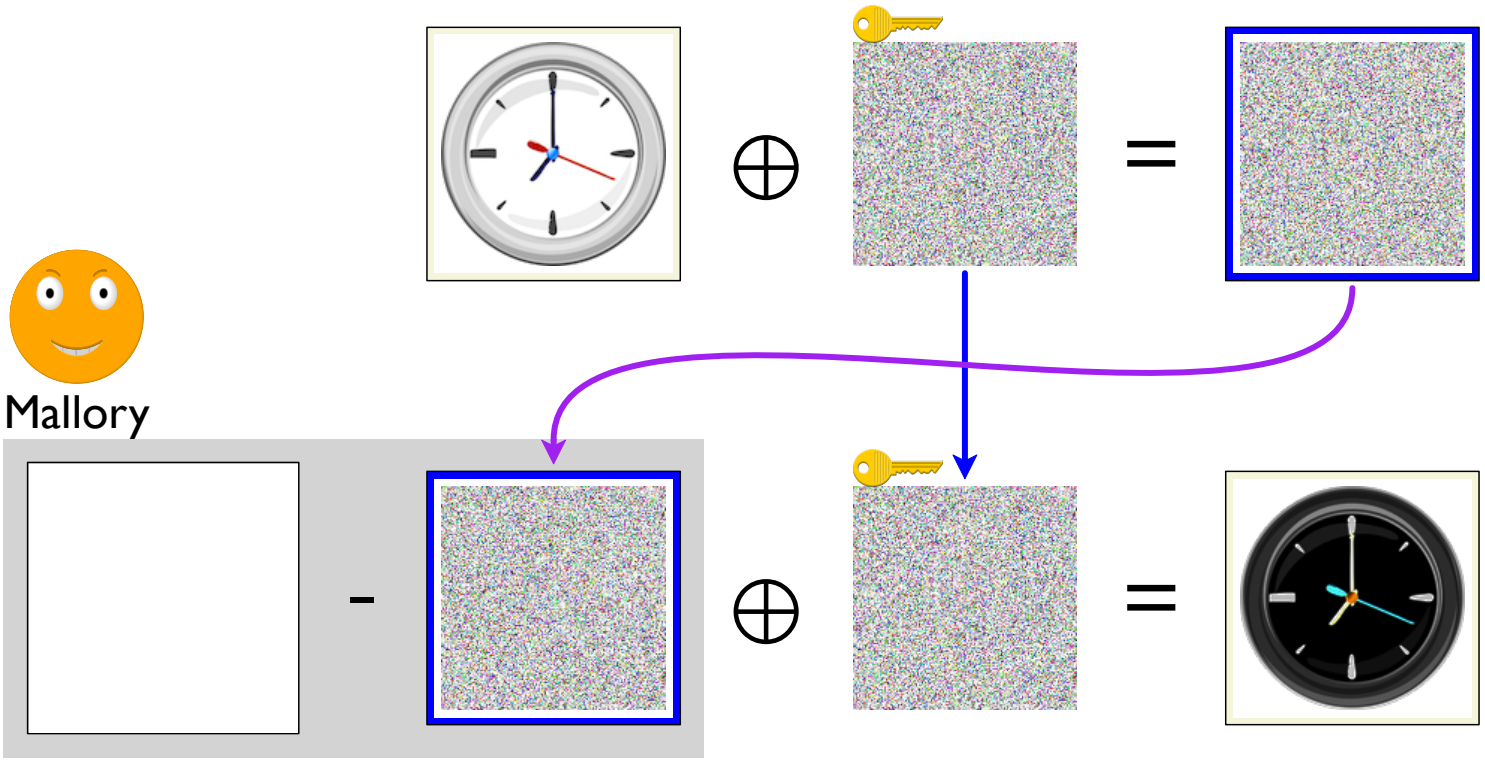
Potentially Correct Use of Stream Cipher and Key



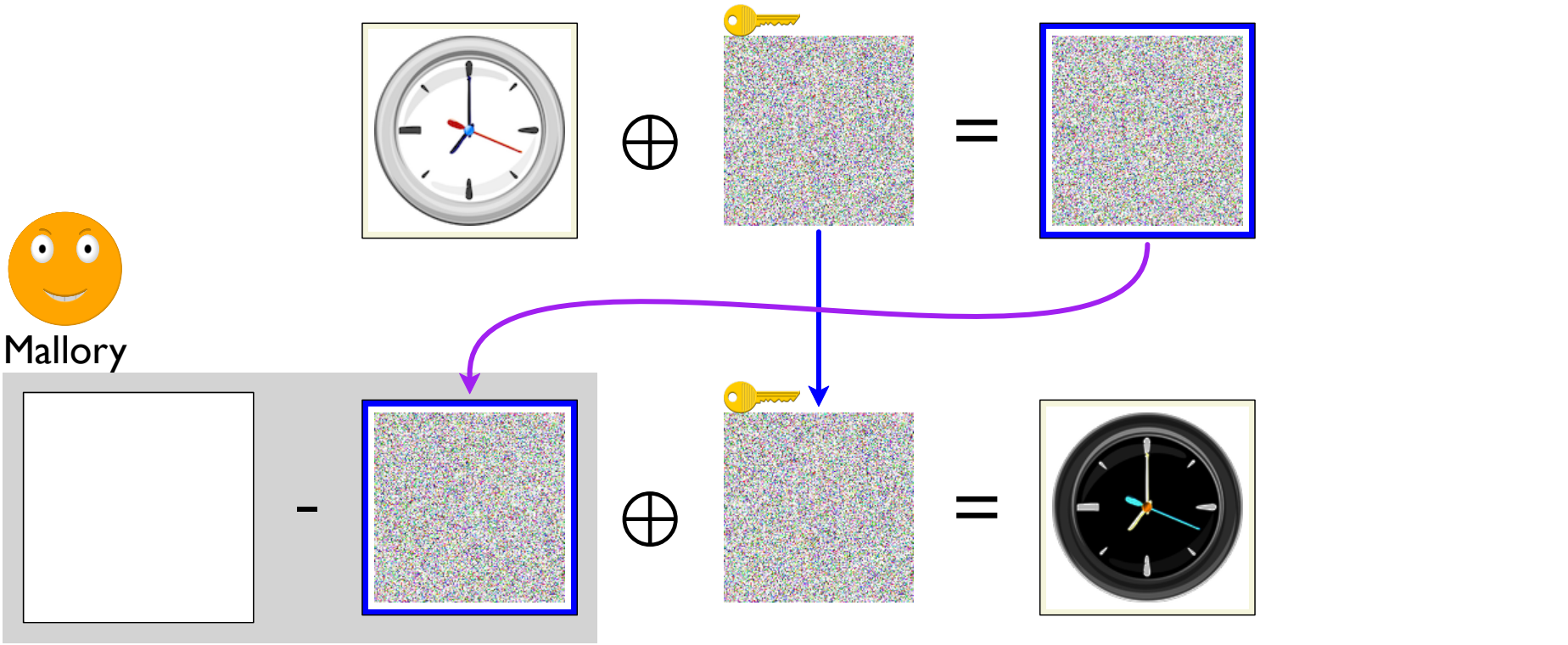
Stream Cipher Lack of Integrity



Stream Cipher Lack of Integrity



Stream Cipher Lack of Integrity



Again, we'll need more ingredients...

Creating a Really Good PRNG

We need a *crytographically secure* PRNG (**CSPRNG**)

- Cannot predict next output from previous outputs
- Passes statistical randomness tests

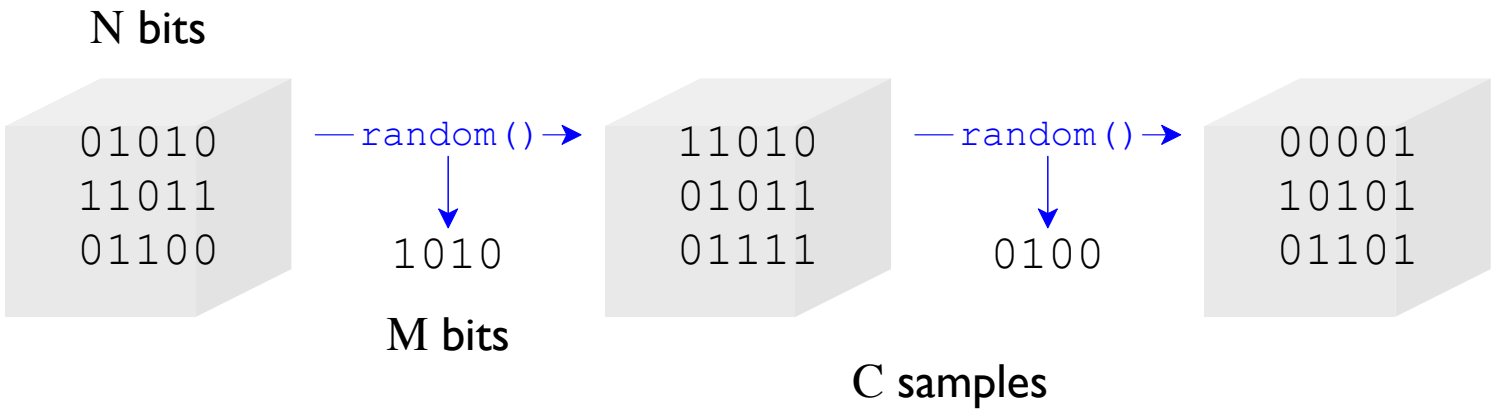
Creating a Really Good PRNG

We need a *crytographically secure* PRNG (**CSPRNG**)

- Cannot predict next output from previous outputs
- Passes statistical randomness tests

- Cannot use current state to predict *previous* output

PRNG

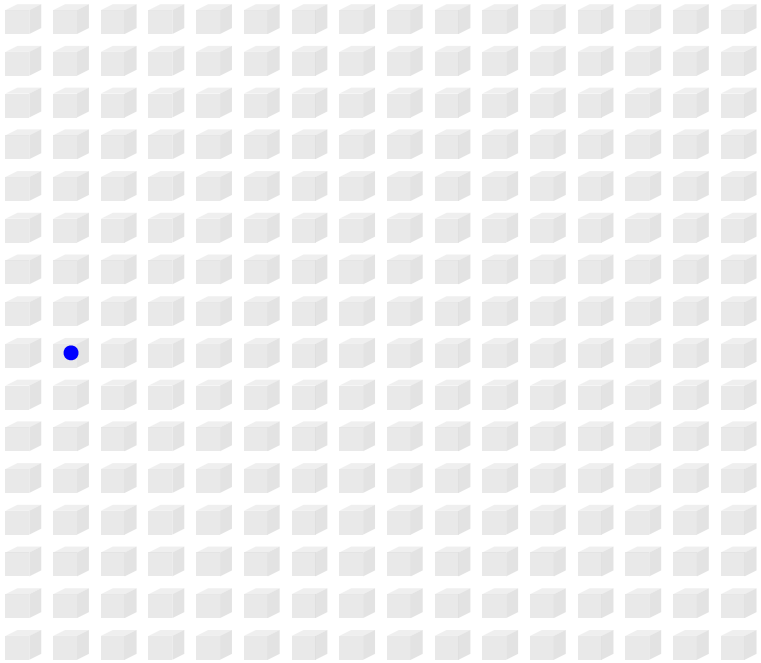


2^N possible states

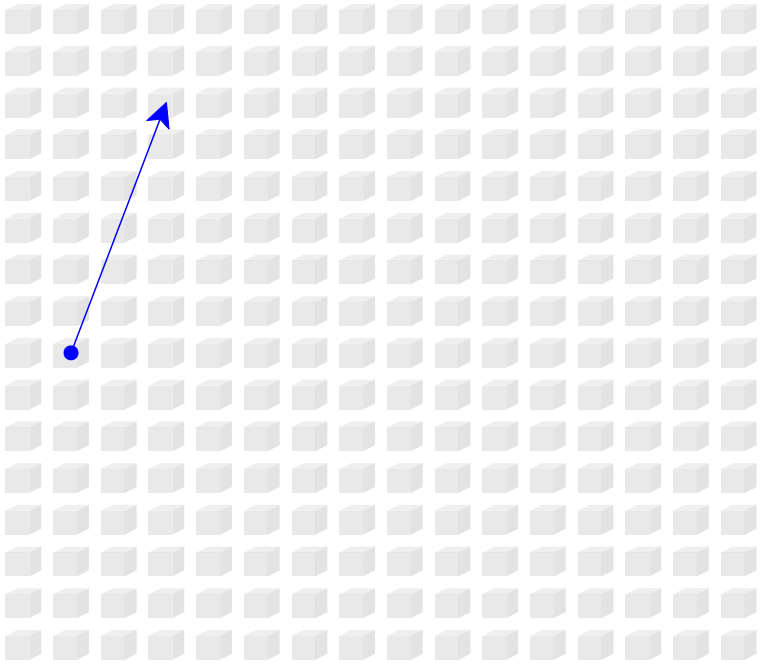
$2^{C \times M}$ possible sequences

$\Rightarrow$ lots of overlap when $C \times M \gg N$

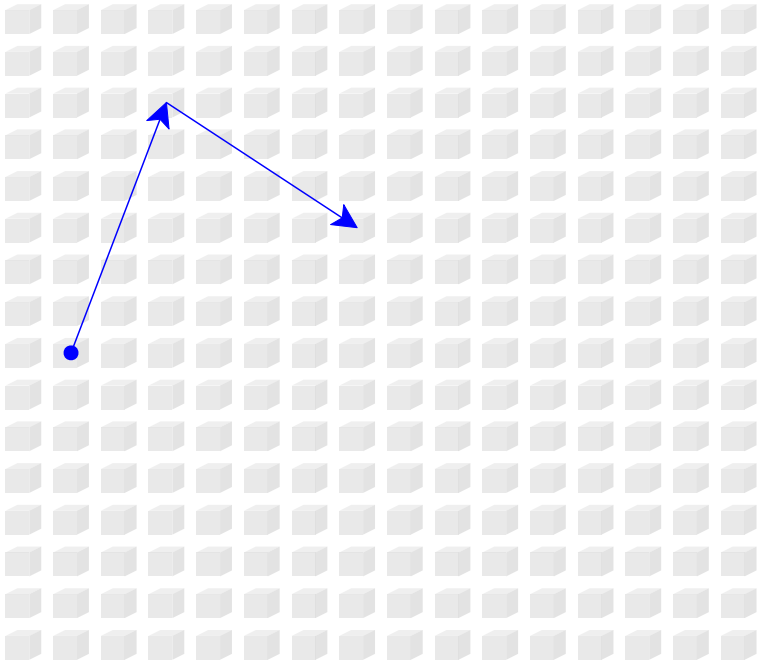
PRNG States



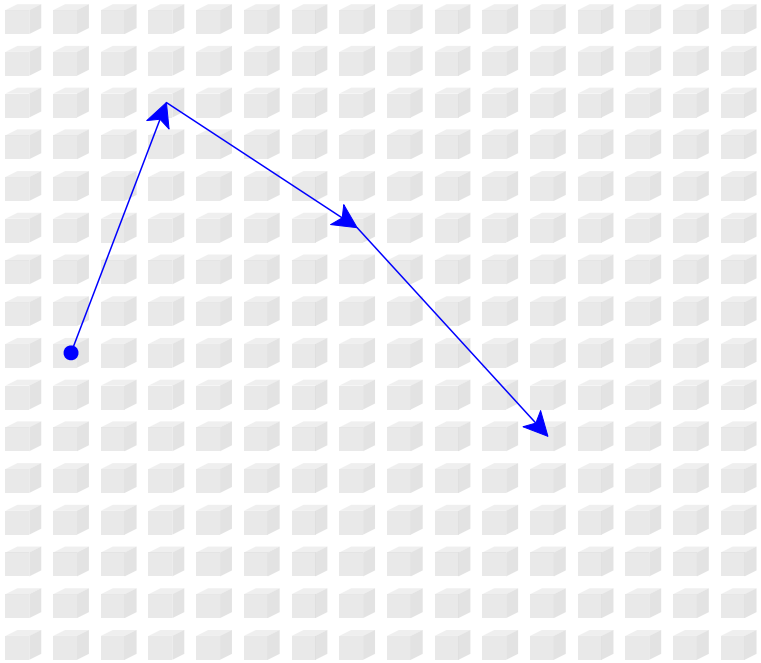
PRNG States



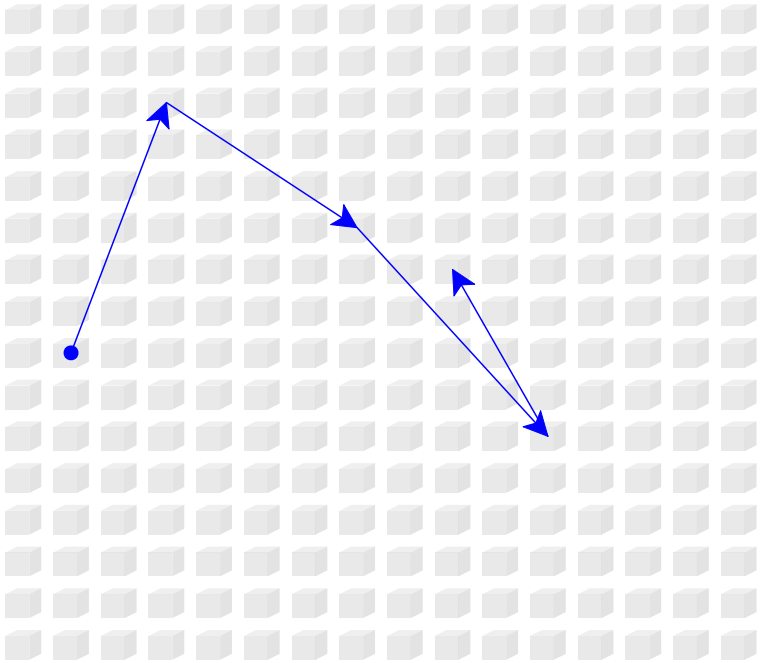
PRNG States



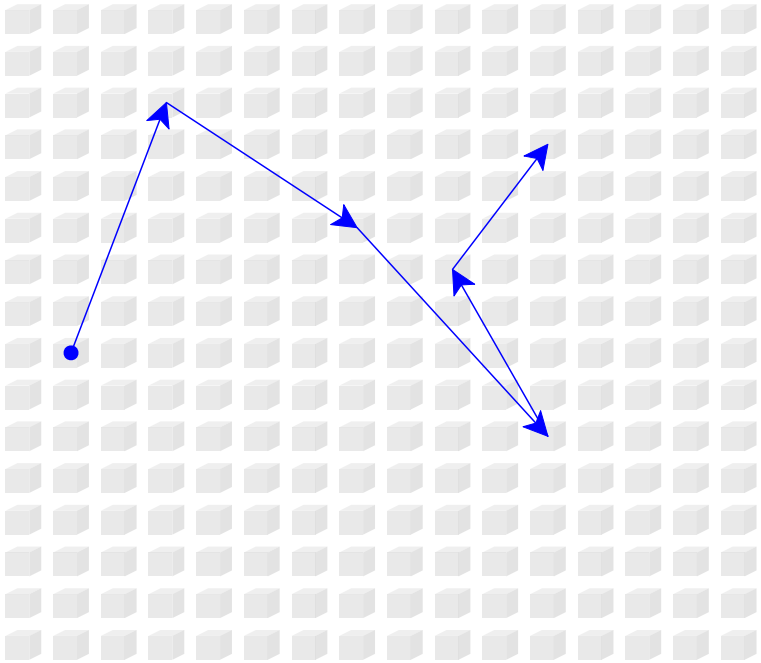
PRNG States



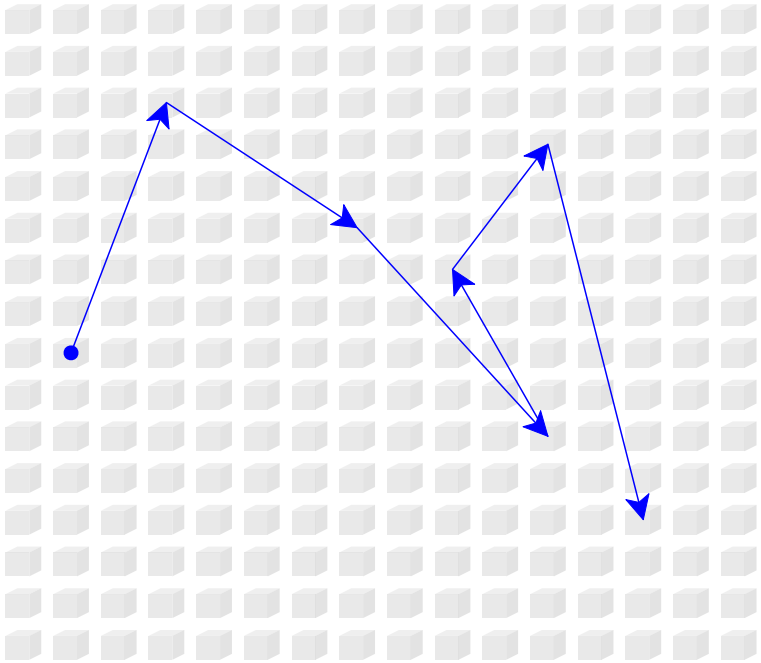
PRNG States



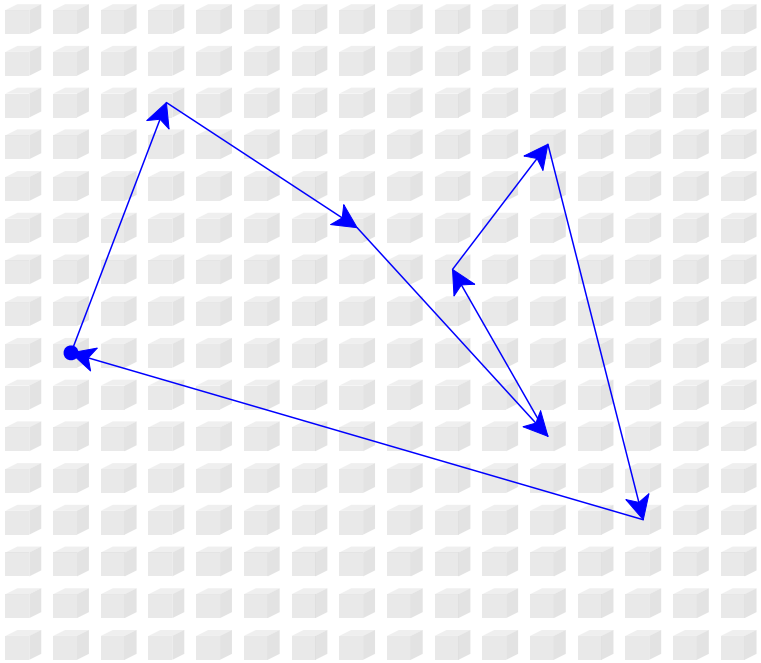
PRNG States



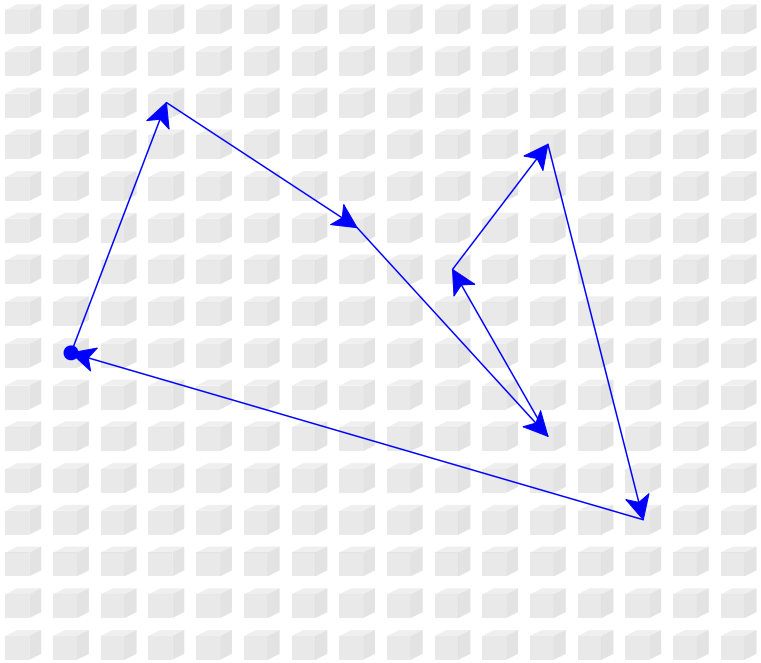
PRNG States



PRNG States

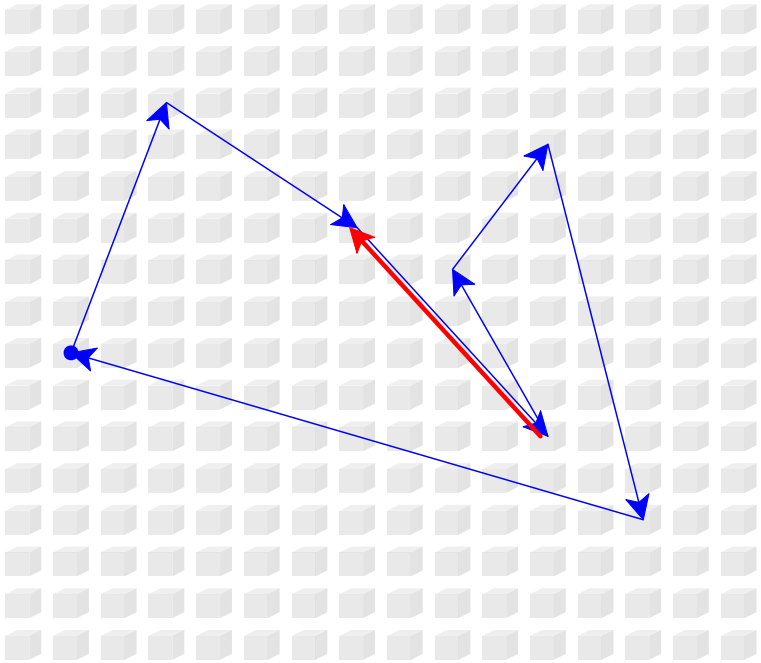


PRNG States



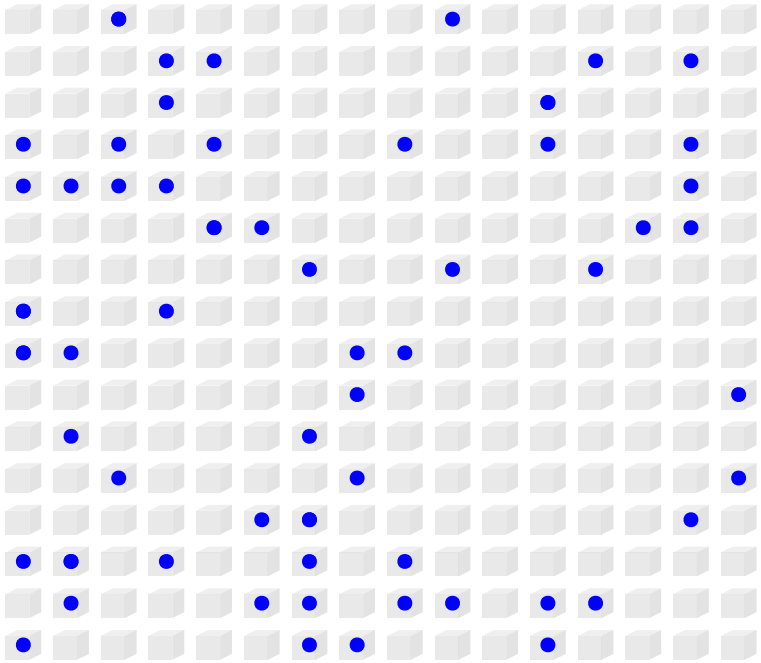
Cycle is inevitable, but we want it to be as long as possible

PRNG States



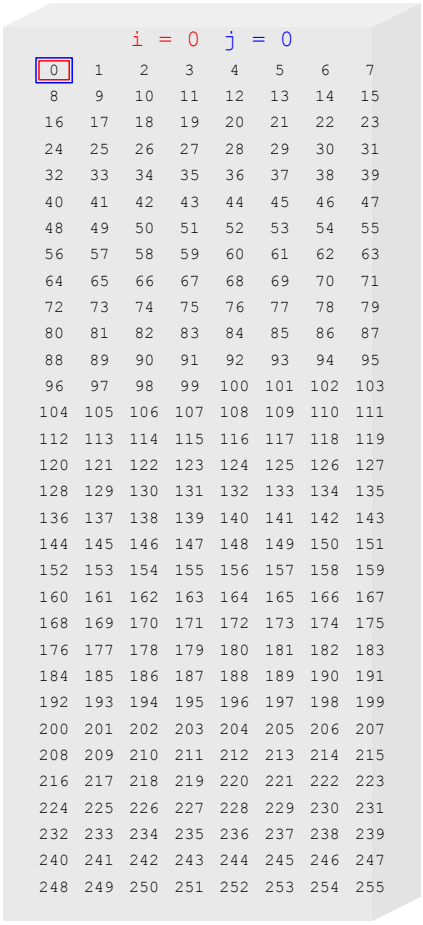
CSPRNG implies that
you can't reverse an arrow
even if you see the state

PRNG States



Key to initialize usually
< N bits, so only
some starts are possible

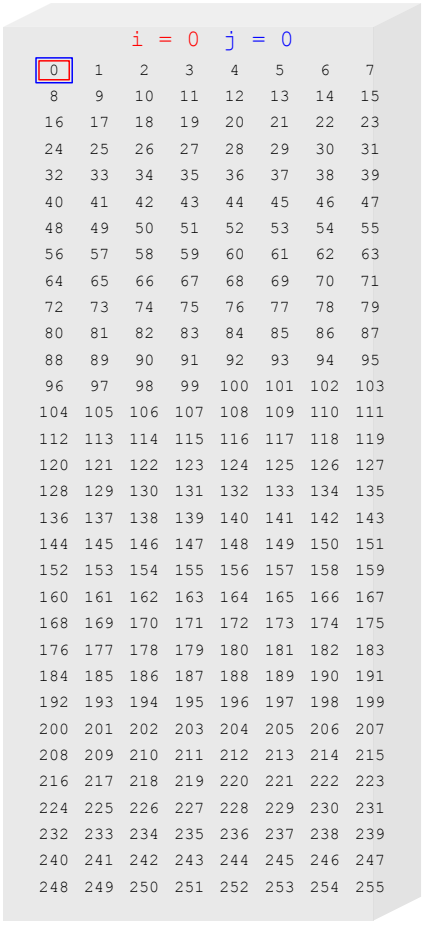
Historical PRNG: RC4



RC4 is “Ron’s Code”

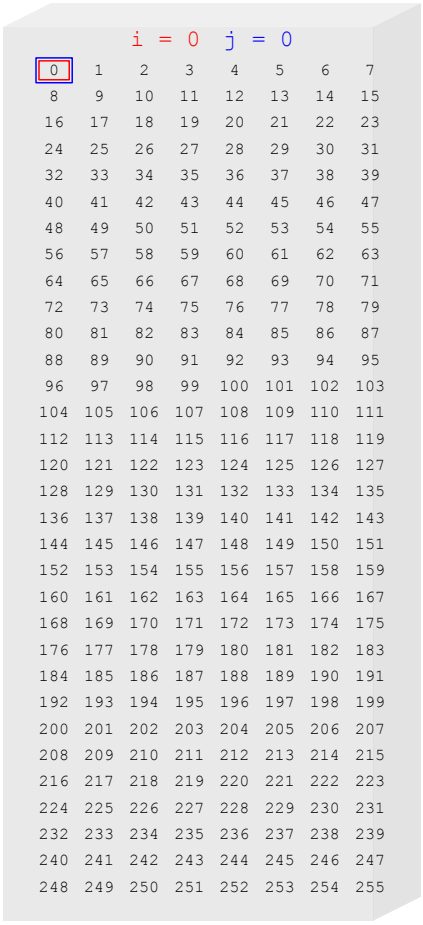
Used in the 1990s, but several vulnerabilities have been found, so don’t use it

Historical PRNG: RC4




```
def S = Array(0..255)
def i = 0
def j = 0
```

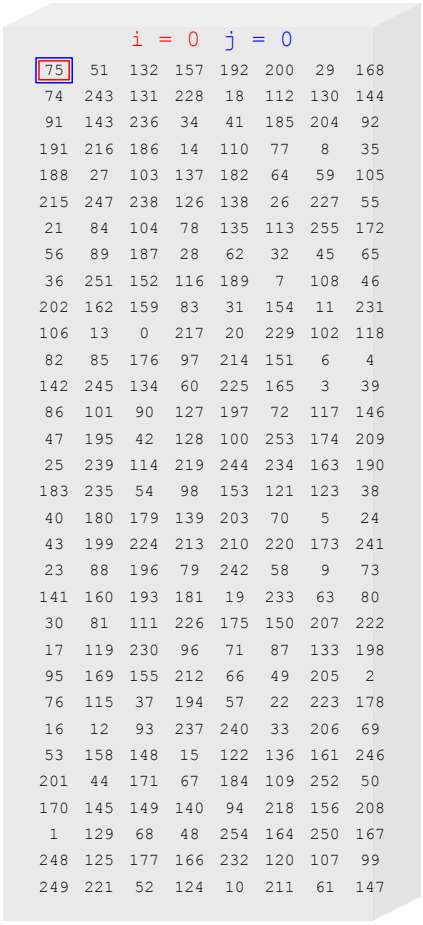
Historical PRNG: RC4




```
j := 0
for (i: 0..255):
  j := (j + S[i] + key[i % key.length]) % 256
  swap(S[i], S[j])

i := 0
j := 0
```

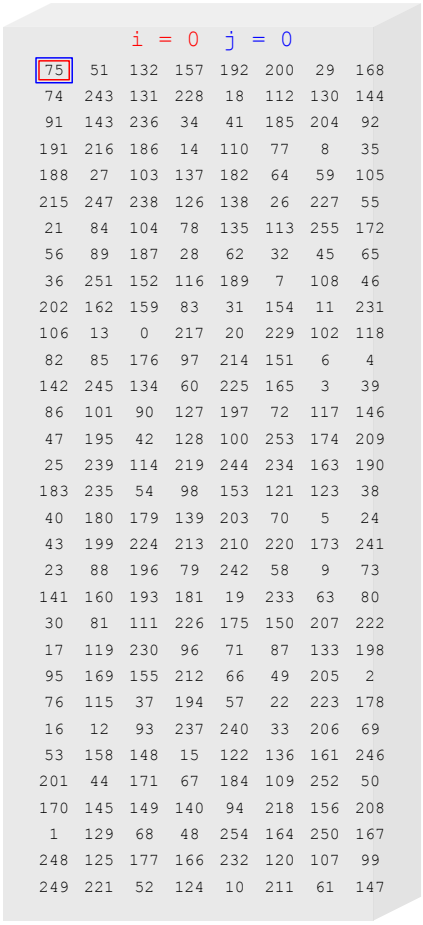
This and later code based on
https://en.wikipedia.org/wiki/RC4

Historical PRNG: RC4




```
def key = "Secret"

j := 0
for (i: 0..255):
  j := (j + S[i] + key[i % key.length]) % 256
  swap(S[i], S[j])

i := 0
j := 0
```

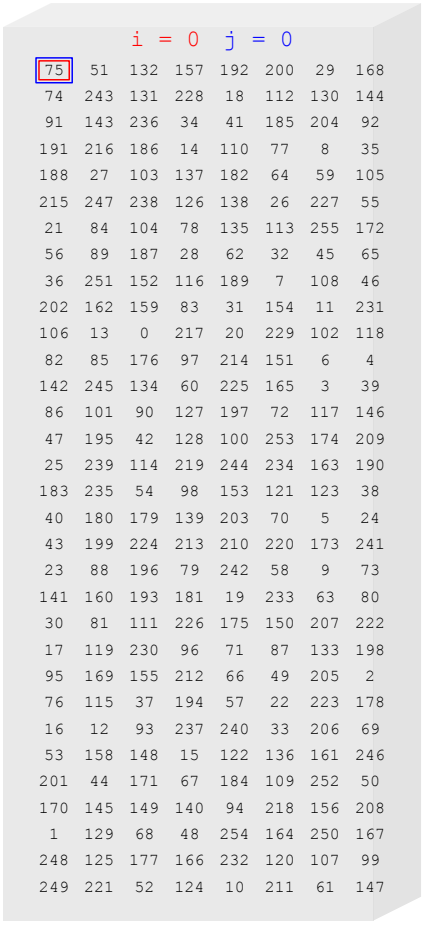
Historical PRNG: RC4



any length, but between 5 and 16 bytes is common

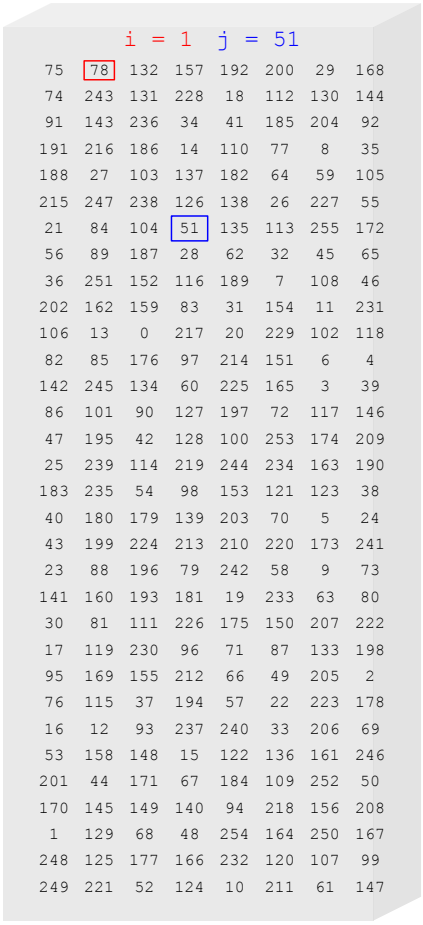
```
def key = "Secret"

j := 0
for (i: 0..255):
  j := (j + S[i] + key[i % key.length]) % 256
  swap(S[i], S[j])

i := 0
j := 0
```

Historical PRNG: RC4




```
def next_byte():
  i := (i+1) mod 256
  j := (j+S[i]) mod 256
  swap(S[i], S[j])
  return S[(S[i] + S[j]) mod 256]
```

Historical PRNG: RC4




```
def next_byte():
  i := (i+1) mod 256
  j := (j+S[i]) mod 256
  swap(S[i], S[j])
  return S[(S[i] + S[j]) mod 256]

next_byte() = 235
```

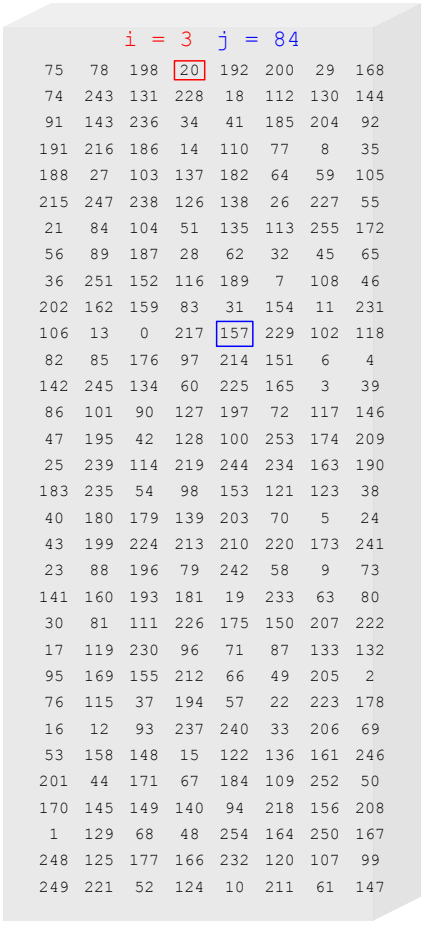
Historical PRNG: RC4

i = 2 j = 183

75	78	198	157	192	200	29	168
74	243	131	228	18	112	130	144
91	143	236	34	41	185	204	92
191	216	186	14	110	77	8	35
188	27	103	137	182	64	59	105
215	247	238	126	138	26	227	55
21	84	104	51	135	113	255	172
56	89	187	28	62	32	45	65
36	251	152	116	189	7	108	46
202	162	159	83	31	154	11	231
106	13	0	217	20	229	102	118
82	85	176	97	214	151	6	4
142	245	134	60	225	165	3	39
86	101	90	127	197	72	117	146
47	195	42	128	100	253	174	209
25	239	114	219	244	234	163	190
183	235	54	98	153	121	123	38
40	180	179	139	203	70	5	24
43	199	224	213	210	220	173	241
23	88	196	79	242	58	9	73
141	160	193	181	19	233	63	80
30	81	111	226	175	150	207	222
17	119	230	96	71	87	133	132
95	169	155	212	66	49	205	2
76	115	37	194	57	22	223	178
16	12	93	237	240	33	206	69
53	158	148	15	122	136	161	246
201	44	171	67	184	109	252	50
170	145	149	140	94	218	156	208
1	129	68	48	254	164	250	167
248	125	177	166	232	120	107	99
249	221	52	124	10	211	61	147

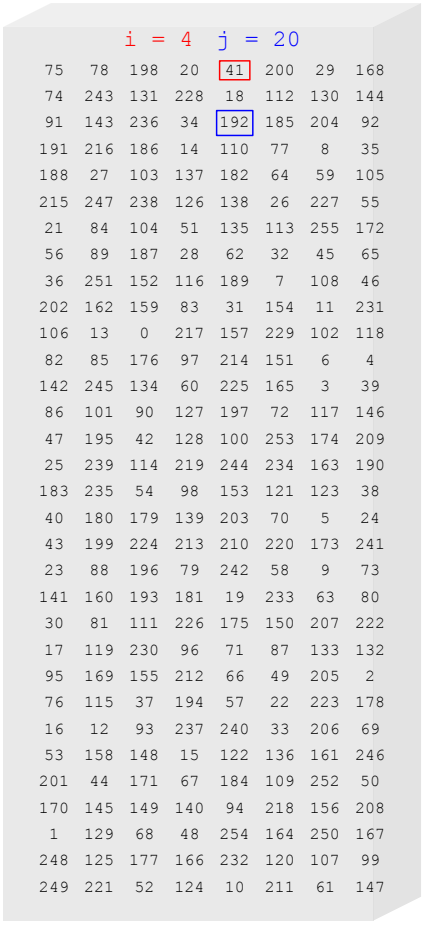
```
def next_byte():
  i := (i+1) mod 256
  j := (j+S[i]) mod 256
  swap(S[i], S[j])
  return S[(S[i] + S[j]) mod 256]

next_byte() = 159
```

Historical PRNG: RC4



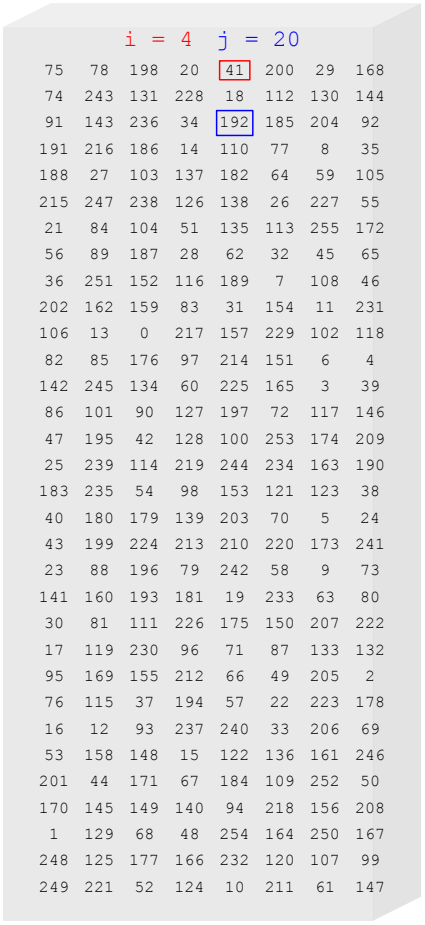

```
def next_byte():
  i := (i+1) mod 256
  j := (j+S[i]) mod 256
  swap(S[i], S[j])
  return S[(S[i] + S[j]) mod 256]

next_byte() = 119
```

Historical PRNG: RC4




```
def next_byte():
  i := (i+1) mod 256
  j := (j+S[i]) mod 256
  swap(S[i], S[j])
  return S[(S[i] + S[j]) mod 256]

next_byte() = 129
```

Historical PRNG: RC4

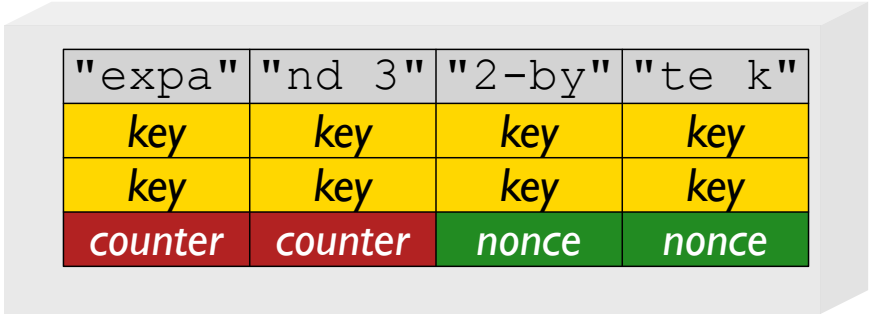


Vulnerabilities include

- first few bytes expose key
- bias in consecutive pairs
- bias toward repeating pairs

Modern PRNG: ChaCha

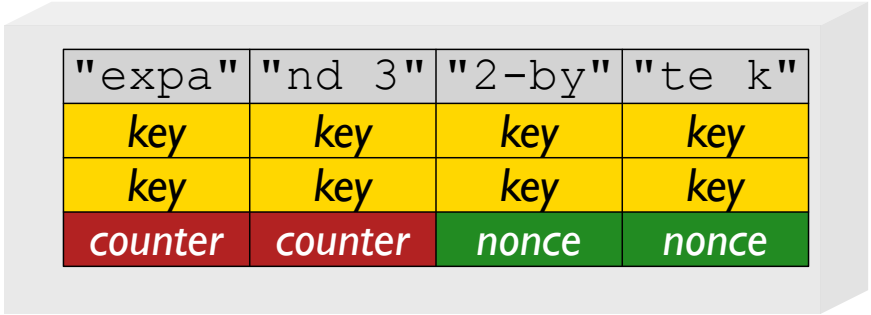
`chacha(`*key*`,` *counter*`,` *nonce*`)` → PRNG



https://en.wikipedia.org/wiki/Salsa20

Modern PRNG: ChaCha

`chacha(`*key*`,` *counter*`,` *nonce*`)` → PRNG



https://en.wikipedia.org/wiki/Salsa20

Applied alternately to columns then diagonals:

```
quarter_round(a, b, c, d):
  a += b; d ⊕= a; d <<<= 16
  c += d; b ⊕= c; b <<<= 12
  a += b; d ⊕= a; d <<<= 8
  c += d; b ⊕= c; b <<<= 7
```

Summary

Stream ciphers use a PRNG plus $\oplus$

Don't reuse a single key

Confidentiality only; need more layers for other goals

Modern CSPRNGs use a mixture of <<< and $\oplus$ on internal state