

# Lecture 7

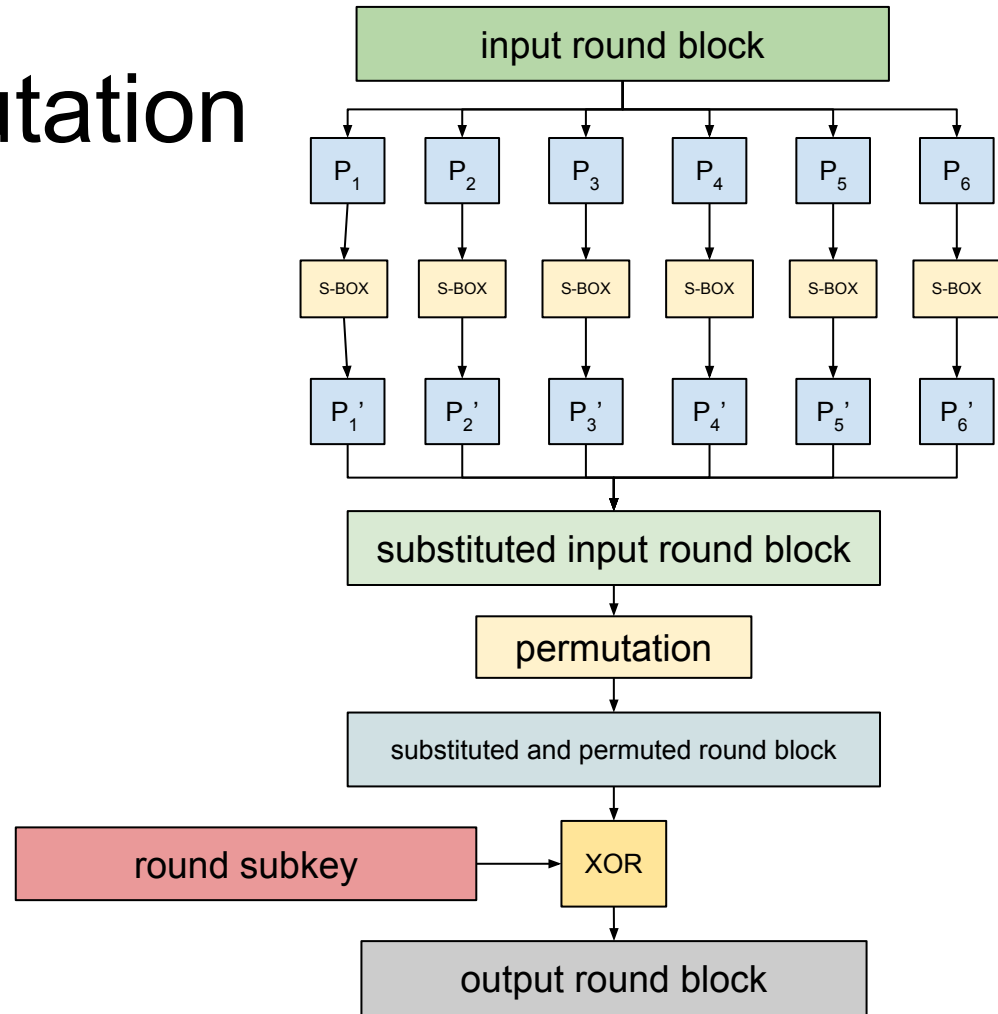
Cryptography 4

# Substitution-permutation ciphers

- Square attack

# Substitution-permutation ciphers

- S-Box
- P-Box
- Combine with key
- Confusion and diffusion

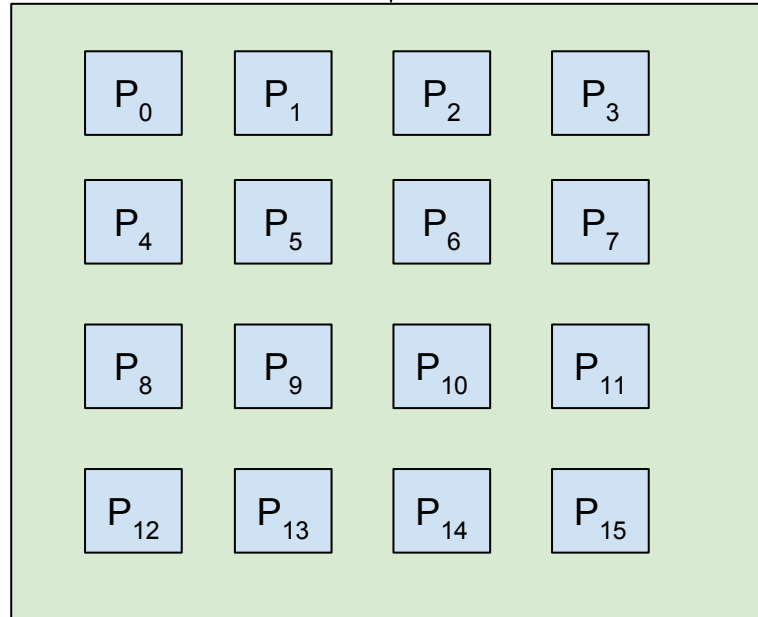


# Square attacks

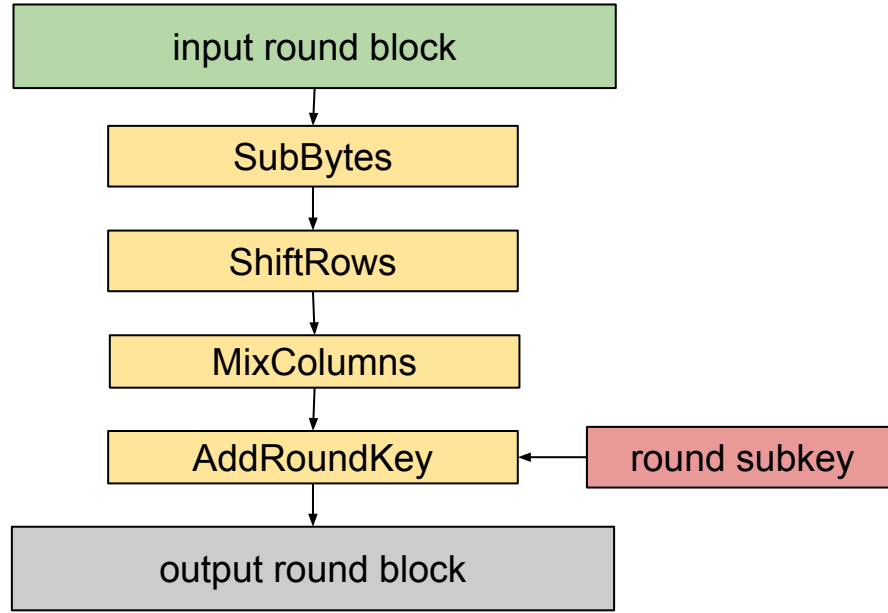
- Integral cryptanalysis
- AES (reduced to 4 rounds (from 10))
  - Chosen plaintext

# Square

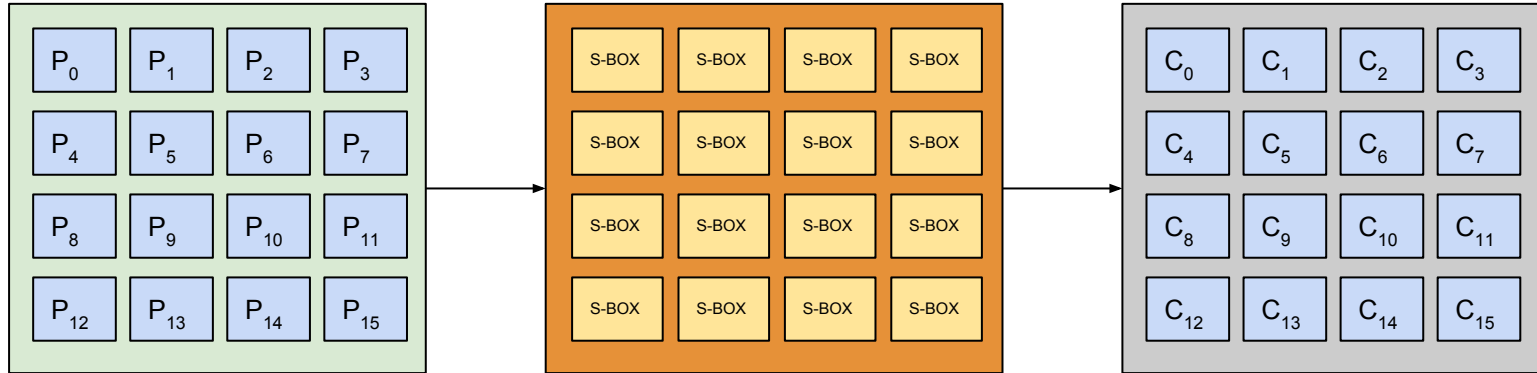
input round block



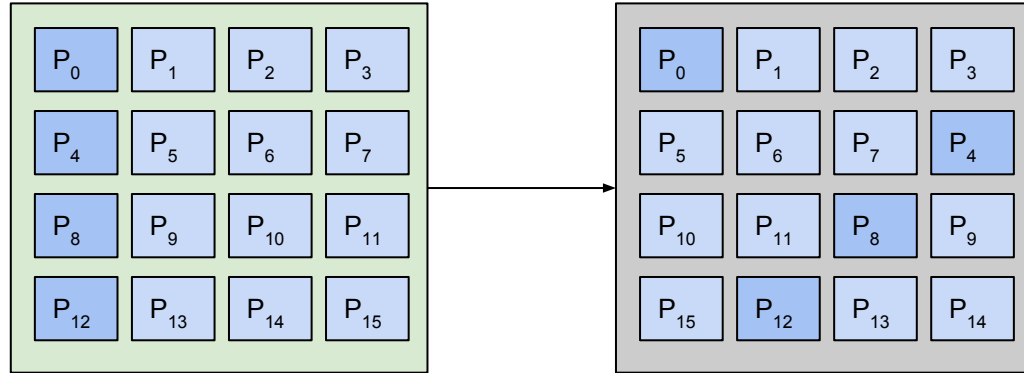
# AES



# SubBytes

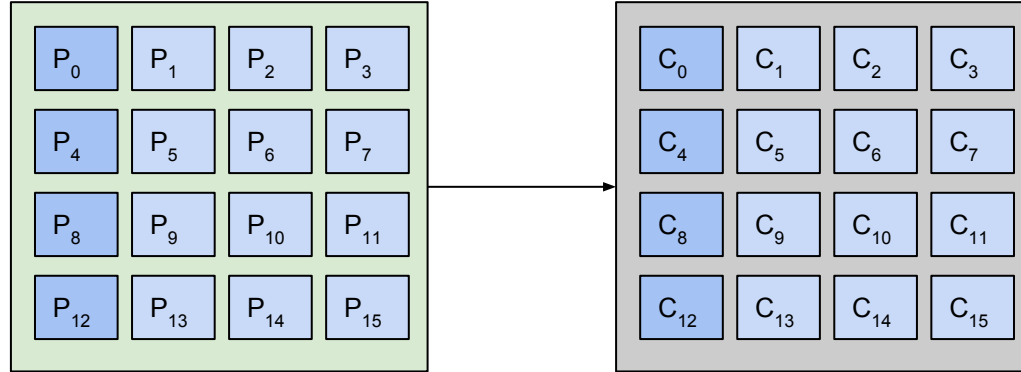


# ShiftRows





# MixColumns



$\text{GF}(2^8)$ ,  $p(x) = P_{12}x^3 + P_8x^2 + P_4x + P_0$ ,  $a(x) = 3x^3 + x^2 + x + 2$ ,  $p(x) * a(x), \text{ mod } x^4 + 1$

$$C_0 = 2P_0 \oplus 3P_4 \oplus P_8 \oplus P_{12}$$

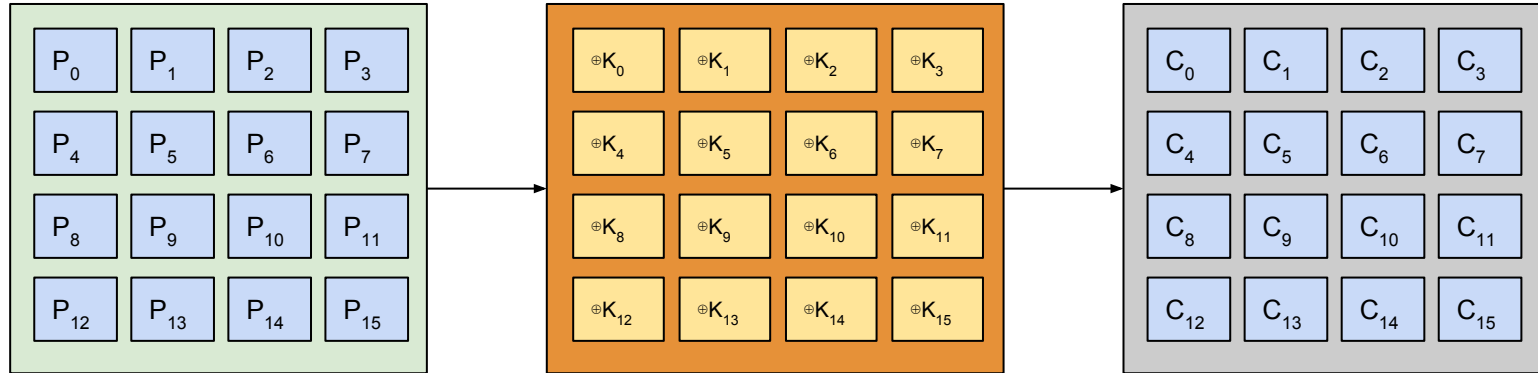
$$C_4 = 2P_4 \oplus 3P_8 \oplus P_{12} \oplus P_0$$

$$C_8 = 2P_8 \oplus 3P_{12} \oplus P_0 \oplus P_4$$

$$C_{12} = 2P_{12} \oplus 3P_0 \oplus P_4 \oplus P_3$$

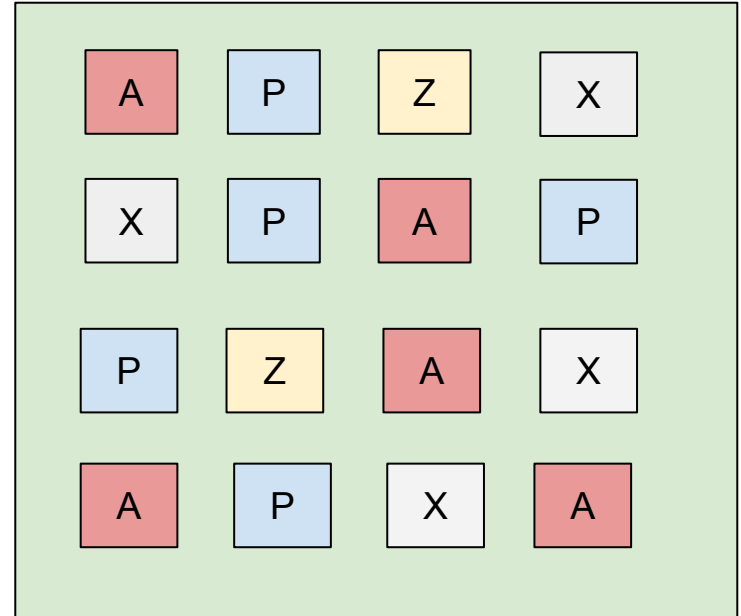
Not present in last round

# AddRoundKey

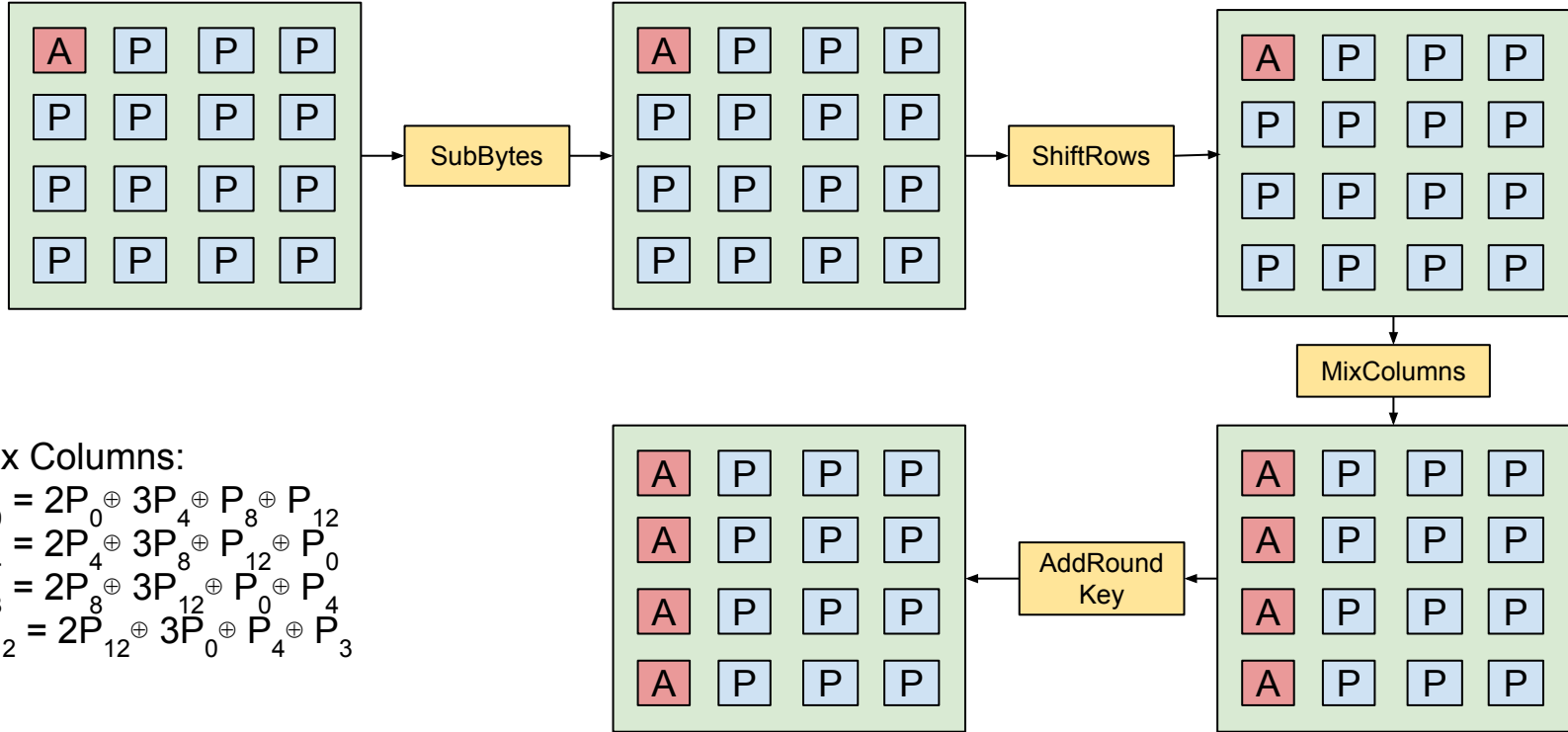


# Active and passive byte states

- Set of plaintext
- Passive state: for each two plaintext the same square element
- Active state: for each two plaintext different square element
- Zero state:  $\oplus$  elements from all plaintext = 0
  - $0 \oplus 1 \oplus \dots \oplus 255 = 0$
- Example:
  - 256 plaintext, first byte: 0,1,2,...,255, other bytes: 0
  - Before first round: top left element is active, other elements passive



# Round 1



Mix Columns:

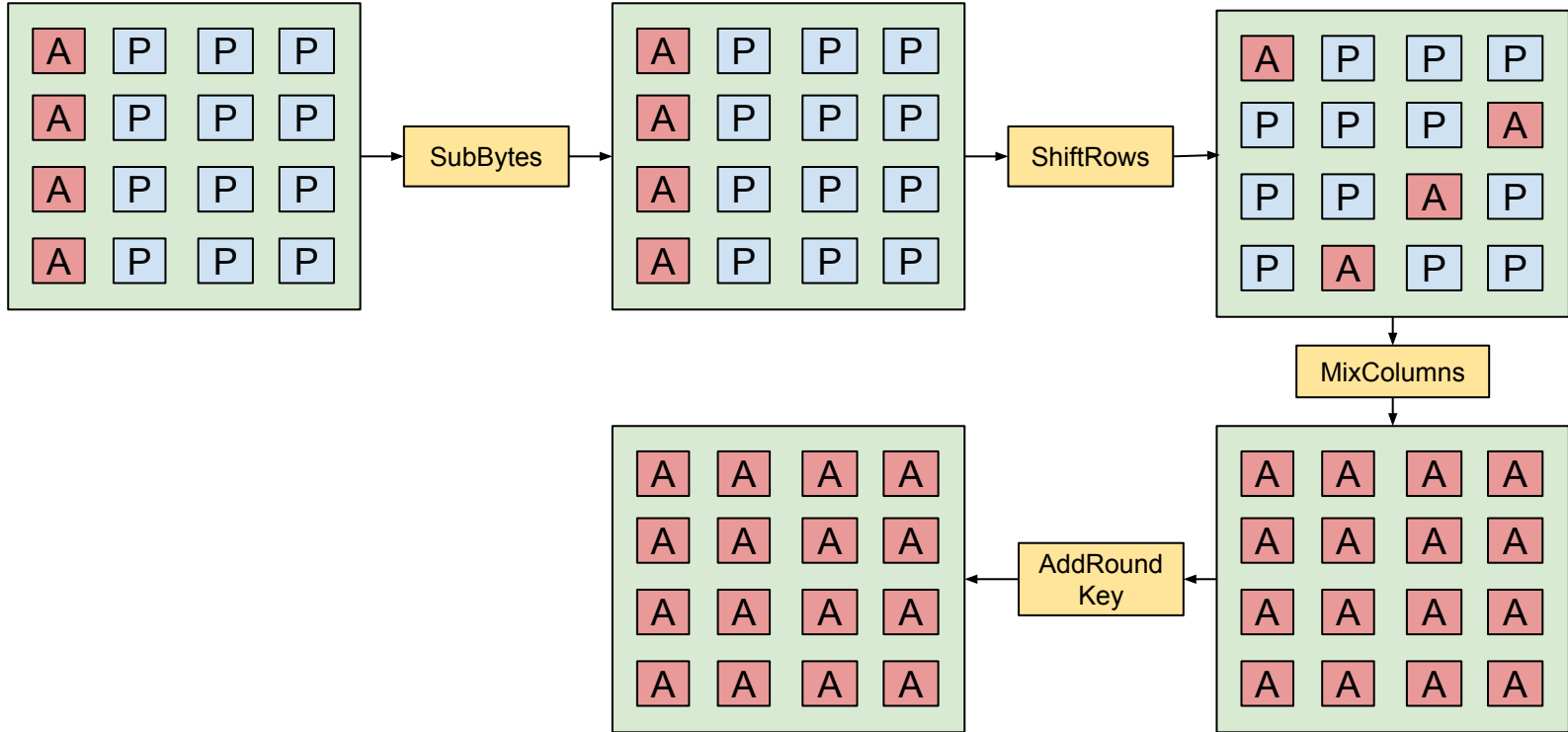
$$C_0 = 2P_0 \oplus 3P_4 \oplus P_8 \oplus P_{12}$$

$$C_4 = 2P_4 \oplus 3P_8 \oplus P_{12} \oplus P_0$$

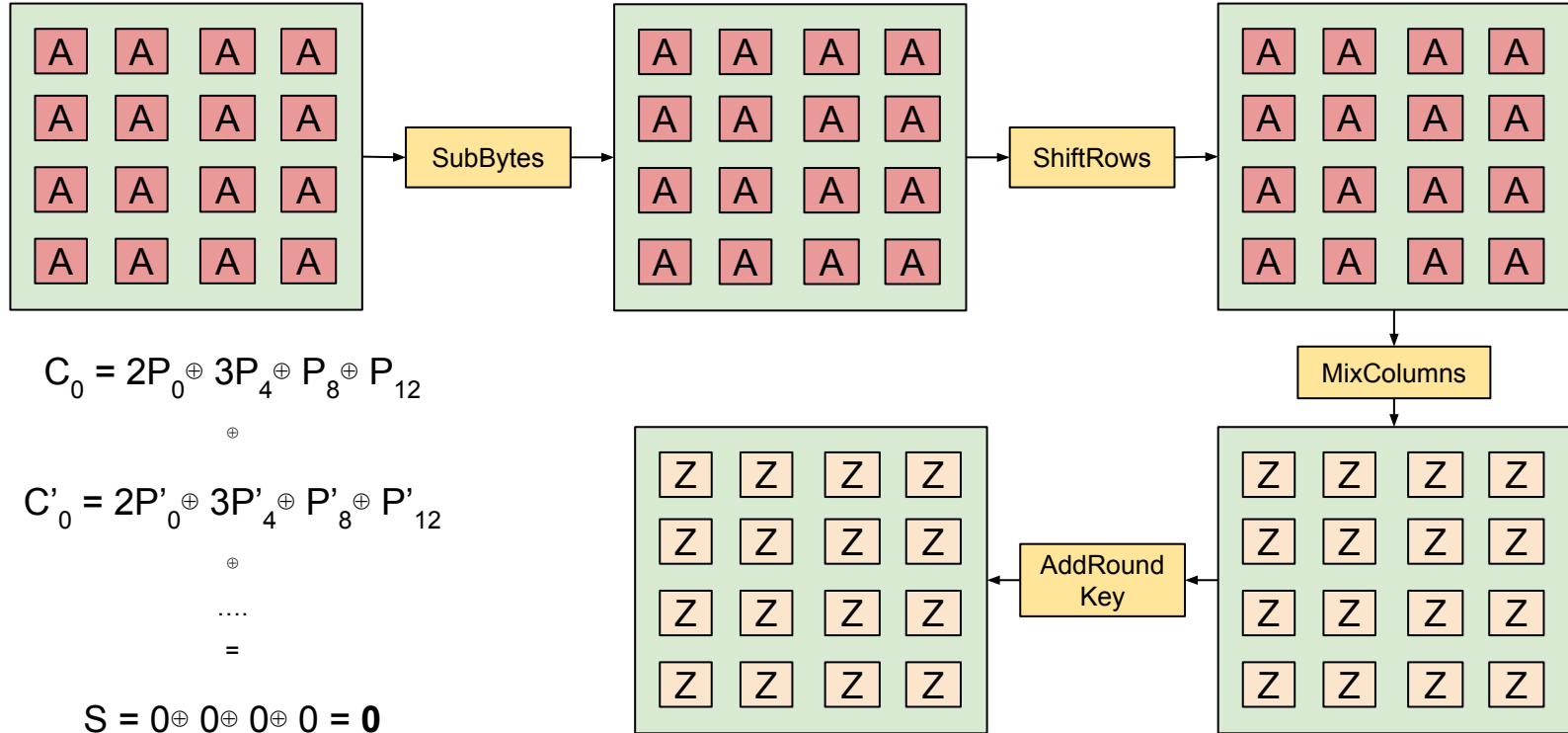
$$C_8 = 2P_8 \oplus 3P_{12} \oplus P_0 \oplus P_4$$

$$C_{12} = 2P_{12} \oplus 3P_0 \oplus P_4 \oplus P_8$$

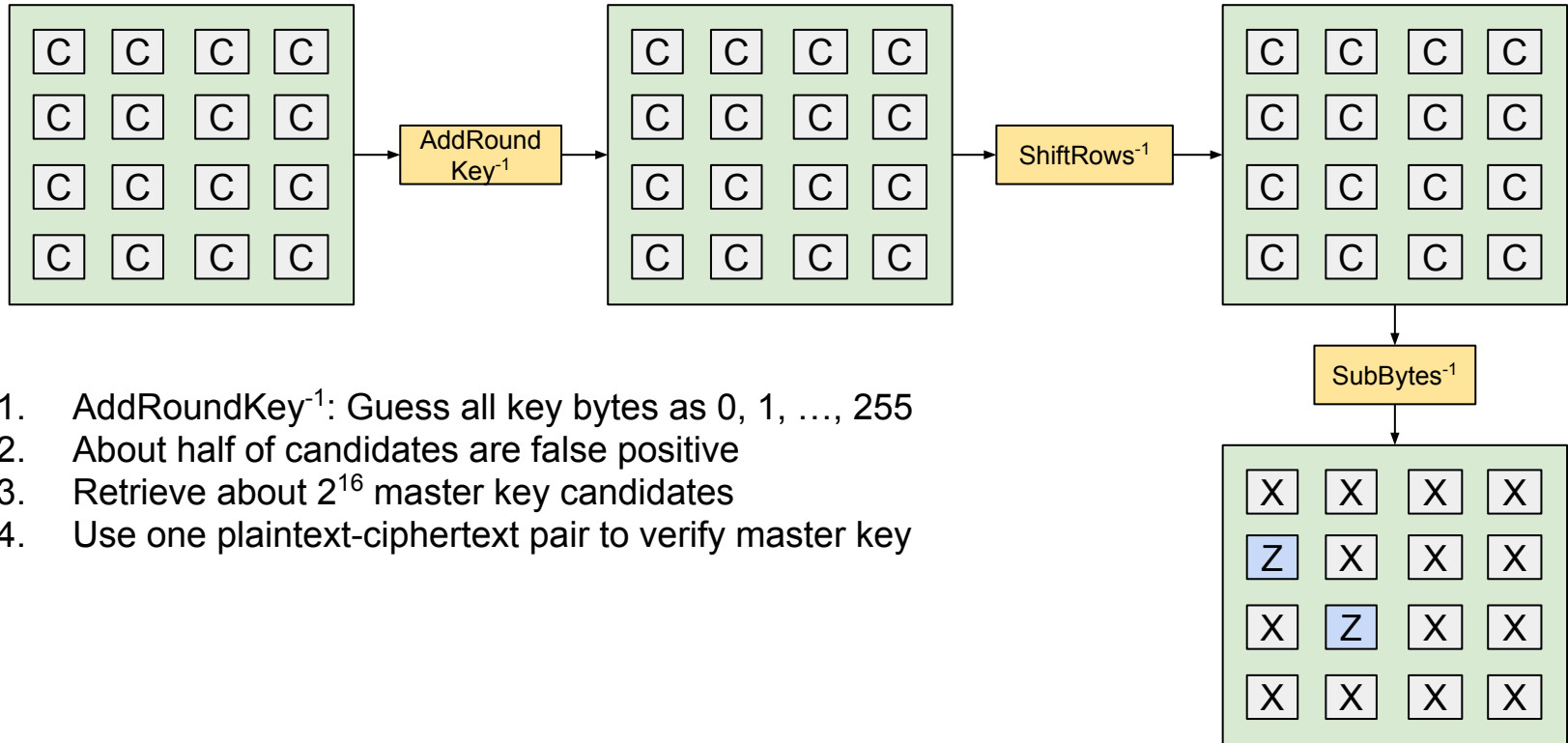
# Round 2



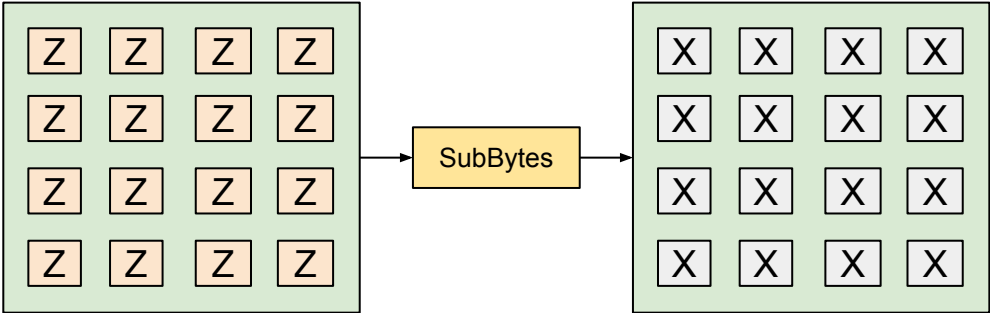
# Round 3



# Key byte guessing



# Round 4





# exercise

Drive: `crypto4/square/`

`aes.py`: aes implementation, bonus method: `master_from_round` -> calculate master key from round key

`client.py`:

`encrypt(data)` -> returns encrypted data (one block)

`get_encrypted_flag()` -> returns encrypted flag

goal: **decrypt flag**

# Pseudorandom generators

# PRNG vs CSPRNG

Self explanatory:

**P**seudo **R**andom **N**umber **G**enerator

**C**ryptographically **S**ecure **P**seudo **R**andom **N**umber **G**enerator

PRNG also known as a **D**eterministic **R**andom **B**it **G**enerator

# PRNGs

Given

- $P$  - a probability distribution on  $(\mathbb{R}, \mathfrak{B})$  (where  $\mathfrak{B}$  is the standard [Borel field](#) on the real line)
- $\mathfrak{F}$  - a non-empty collection of Borel sets  $\mathfrak{F} \subseteq \mathfrak{B}$ , e.g.  $\mathfrak{F} = \{(-\infty, t] : t \in \mathbb{R}\}$ . If  $\mathfrak{F}$  is not specified, it may be either  $\mathfrak{B}$  or  $\{(-\infty, t] : t \in \mathbb{R}\}$ , depending on context.
- $A \subseteq \mathbb{R}$  - a non-empty set (not necessarily a Borel set). Often  $A$  is a set between  $P$ 's [support](#) and its [interior](#); for instance, if  $P$  is the uniform distribution on the interval  $(0, 1]$ ,  $A$  might be  $(0, 1]$ . If  $A$  is not specified, it is assumed to be some set contained in the support of  $P$  and containing its interior, depending on context.

We call a function  $f : \mathbb{N}_1 \rightarrow \mathbb{R}$  (where  $\mathbb{N}_1 = \{1, 2, 3, \dots\}$  is the set of positive integers) a **pseudo-random number generator for  $P$  given  $\mathfrak{F}$  taking values in  $A$**  iff

- $f(\mathbb{N}_1) \subseteq A$
- $\forall E \in \mathfrak{F} \quad \forall 0 < \varepsilon \in \mathbb{R} \quad \exists N \in \mathbb{N}_1 \quad \forall N \leq n \in \mathbb{N}_1, \quad \left| \frac{\#\{i \in \{1, 2, \dots, n\} : f(i) \in E\}}{n} - P(E) \right| < \varepsilon$

( $\#S$  denotes the number of elements in the finite set  $S$ .)

It can be shown that if  $f$  is a pseudo-random number generator for the uniform distribution on  $(0, 1)$  and if  $F$  is the [CDF](#) of some given probability distribution  $P$ , then  $F^* \circ f$  is a pseudo-random number generator for  $P$ , where  $F^* : (0, 1) \rightarrow \mathbb{R}$  is the percentile of  $P$ , i.e.  $F^*(x) := \inf\{t \in \mathbb{R} : x \leq F(t)\}$ . Intuitively, an arbitrary distribution can be simulated from a simulation of the standard uniform distribution.

# PRNGs

- *"Algorithm for generating a sequence of numbers whose properties approximate the properties of sequences of random numbers"*
- Perfect PRNG = indistinguishable from random sequence
- Trivia: with N bits of internal state, PRNG will repeat itself every  $2^N$  steps.
- PRNG test: DieHard, DieHarder. For example:
  - The same number of 1s and 0s
  - Distribution of ngrams
  - Spacings between random points
  - Computing pi

# PRNGs: general form

```
class prng_generic:
    def __init__(self, seed):
        self.state = self.initial_state(seed)

    def next(self):
        self.update_state()
        return self.current()
```

# PRNGs: middle-square

- middle-square method: one of first methods of generating pseudorandom numbers
- Invented and used by John von Neumann

-                   1234567890 \*\*2  
1524157875019052100

next(1234567890) = 1578750190

# PRNGs: middle-square

```
class prng_middle_square:  
    def __init__(self, seed):  
        self.state = seed  
  
    def next(self):  
        m = '0'*10 + str(self.state**2)  
        self.state = int(m[-20:][5:15])  
        return self.state
```



# PRNGs: LCG

- **L**inear **C**ongruential **G**enerator
- One of oldest and most popular PRNGs
- Fast, very easy to implement
- Used by old Visual Studio C++, Visual Basic <= 6, Delphi, Pascal, glibc (sometimes), Borland C++, java.util.Random
  
- `state = (state*m + c) % n`

`next(1234567890) = 804752948`

(with `m=1103515245`, `c=12345`, `n=2**31-1=2147483647`)

# PRNGs: LCG

```
class prng_lcg:
    def __init__(self, seed):
        self.state = seed

    def next(self):
        self.state = (self.state * m + c) % n
        return self.state
```

# PRNGs: Mersenne Twister

- **The** most popular PRNG?
- Fast, rather simple
- Standard implementation is based on  $2^{19937}-1$
- Used by Python, Ruby, PHP, Visual Studio C++, GMP, Common Lisp, Free Pascal, Sage, Excel...
- [https://en.wikipedia.org/wiki/Mersenne\\_Twister#Python\\_implementation](https://en.wikipedia.org/wiki/Mersenne_Twister#Python_implementation)

# Linear Feedback Shift Registers

$$\text{state} = (\text{state} \ll 13) \wedge \text{state}$$

- Useful for pseudo-random numbers generation
- Reversible transformation
- Used in Mersenne Twister

# PRNGs: Mersenne Twister

```
class prng_mt:
    def __init__(self, seed):
        self.index = 624
        self.mt = [0] * 624
        self.mt[0] = seed
        for i in range(1, 624):
            self.mt[i] = 0xffffffff & (1812433253 *
                (self.mt[i - 1] ^ self.mt[i - 1] >> 30) + i)
```

(code ~~stolen~~borrowed from wikipedia)

# PRNGs: Mersenne Twister

```
def next(self):
    if self.index >= 624:
        self.twist()

    y = self.mt[self.index]
    y = y ^ y >> 11
    y = y ^ y << 7 & 2636928640
    y = y ^ y << 15 & 4022730752
    y = y ^ y >> 18
    self.index = self.index + 1
    return y & 0xFFFFFFFF
```

# PRNGs: Mersenne Twister

```
def twist(self):
    for i in range(624):
        y = 0xFFFFFFFF & ((self.mt[i] & 0x80000000) +
                           (self.mt[(i + 1) % 624] & 0x7fffffff))
        self.mt[i] = self.mt[(i + 397) % 624] ^ y >> 1

        if y % 2 != 0:
            self.mt[i] = self.mt[i] ^ 0x9908b0df
    self.index = 0
```

# PRNGs: Putting *pseudo* back in *PRNG*

<https://uw2017.p4.team/lcg1-easy>

(8+1 levels)



# CSPRNGs

Like PRNGs... But **Cryptographically Secure**

Next Bit Test:

Given  $k$  bits of output, adversary can't predict next bit with probability  $> 50\%$

State Compromise Extension Resistance:

After state exposed, impossible to predict previous numbers.

# PRNG vs CSPRNG

	Next Bit Test	State Extension
LCG	X	X
Mersenne Twister	X	X

# CSPRNG design

Designed "from the ground up":

- Blum Blum Shub (quadratic residuosity problem)
- Blum-Micali (discrete logarithm problem)

Based on crypto primitives:

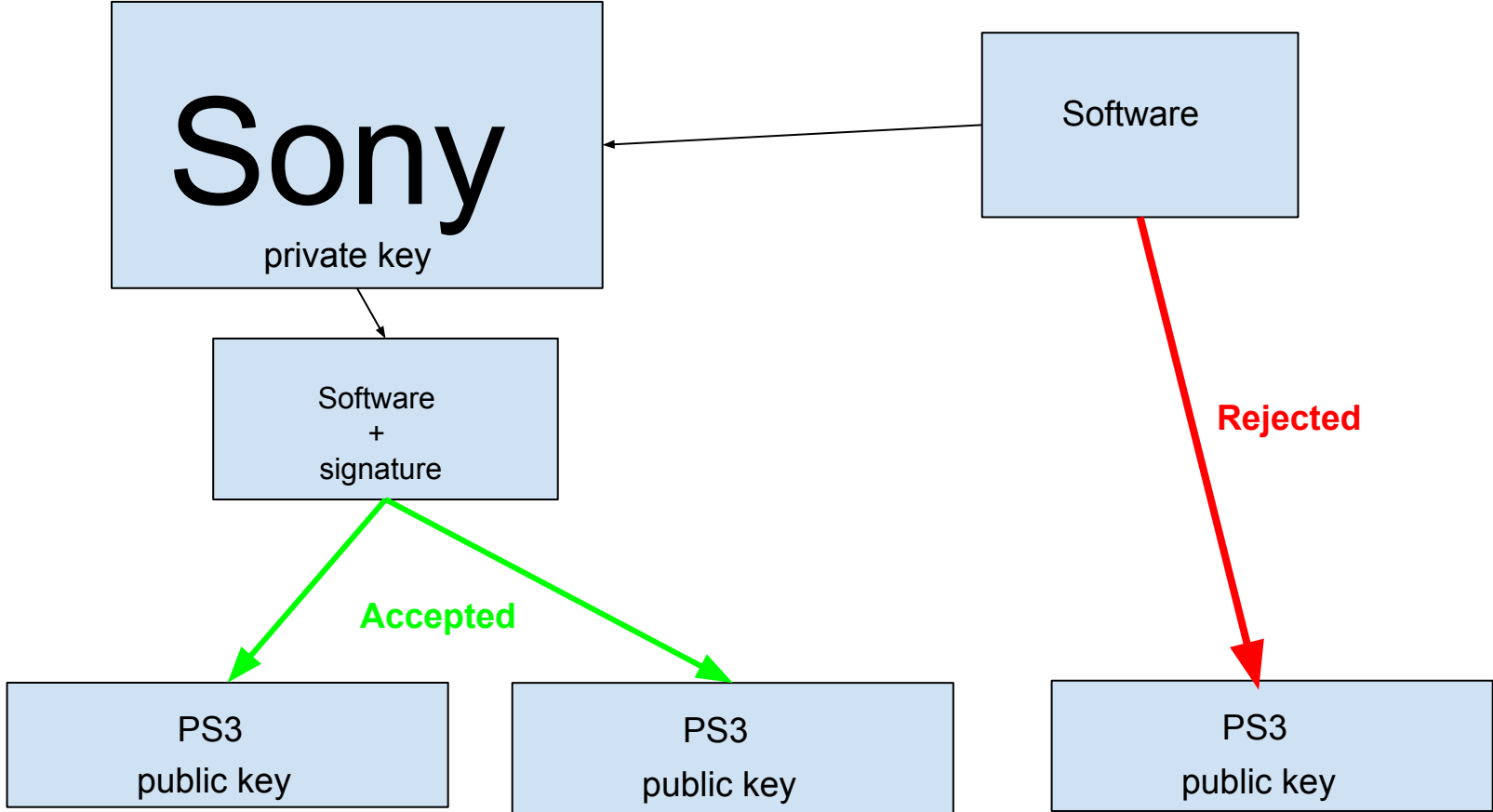
- Block Cipher → CSPRNG
- Stream Cipher → CSPRNG
- Secure Hash → CSPRNG

# Sony crypto fail

## Sony's ECDSA code

```
int getRandomNumber()  
{  
    return 4; // chosen by fair dice roll.  
             // guaranteed to be random.  
}
```

- Team fail0verflow
  - Completely broken Playstation 3 console
- Recovered private key which wasn't present on device
- Bug in Sony ECDSA signing implementation



# DSA algorithm

Parameters:

Numbers P, Q, G

Hash Function H

Private Key:

random X:  $0 < X < Q$

Public Key:

Y:  $Y = G^X \bmod P$

Singing: M - message

**Choose random K:  $0 < K < Q$**

Calc: R:  $R = (G^K \bmod P) \bmod Q$

Calc: S:  $S = K^{-1}(H(M) + X * R) \bmod Q$

Signature: (R, S)

Verification: M - message, (R, S) - signature

$W = S^{-1} \bmod Q$

$V = (G^{H(M)*W \bmod Q} * Y^{R*W \bmod Q} \bmod P) \bmod Q$

Correct if  $V = R$

# Repeated K

Singing: M - message

**Choose random K:  $0 < K < Q$**

Calc: R:  $R = (G^K \bmod P) \bmod Q$

Calc: S:  $S = K^{-1}(H(M)+X*R) \bmod Q$

Signature: (R, S)

- Two signed messages: M, M'
- Signatures: (R, S), (R', S')
- $K = K'$
- $R = R'$
- $S - S' = (K^{-1}(H(M)+X*R) - K^{-1}(H(M')+X*R)) \bmod Q = K^{-1} * (H(M) - H(M')) \bmod Q$
- $K = (H(M) - H(M')) * (S - S')^{-1} \bmod Q$
- $X = (S * K - H(M)) * R^{-1} \bmod Q$



# exercise

Drive: crypto4/dsa/

client.py:

verify(data, signature) - check if signature is correct for data

verify("flag", signature) - if signature is correct then flag is returned

sigs.txt:

msg, signature pairs for 50 messages generated by gen.py

# Bibliography

Joan Daemen and Vincent Rijmen, "AES Proposal: Rijndael"

FIPS PUB 186-4: Digital Signature Standard (DSS), July 2013