ELTE
EÖTVÖS LORÁND
UNIVERSITY

CRYPTOGRAPHY AND SECURITY

Practice

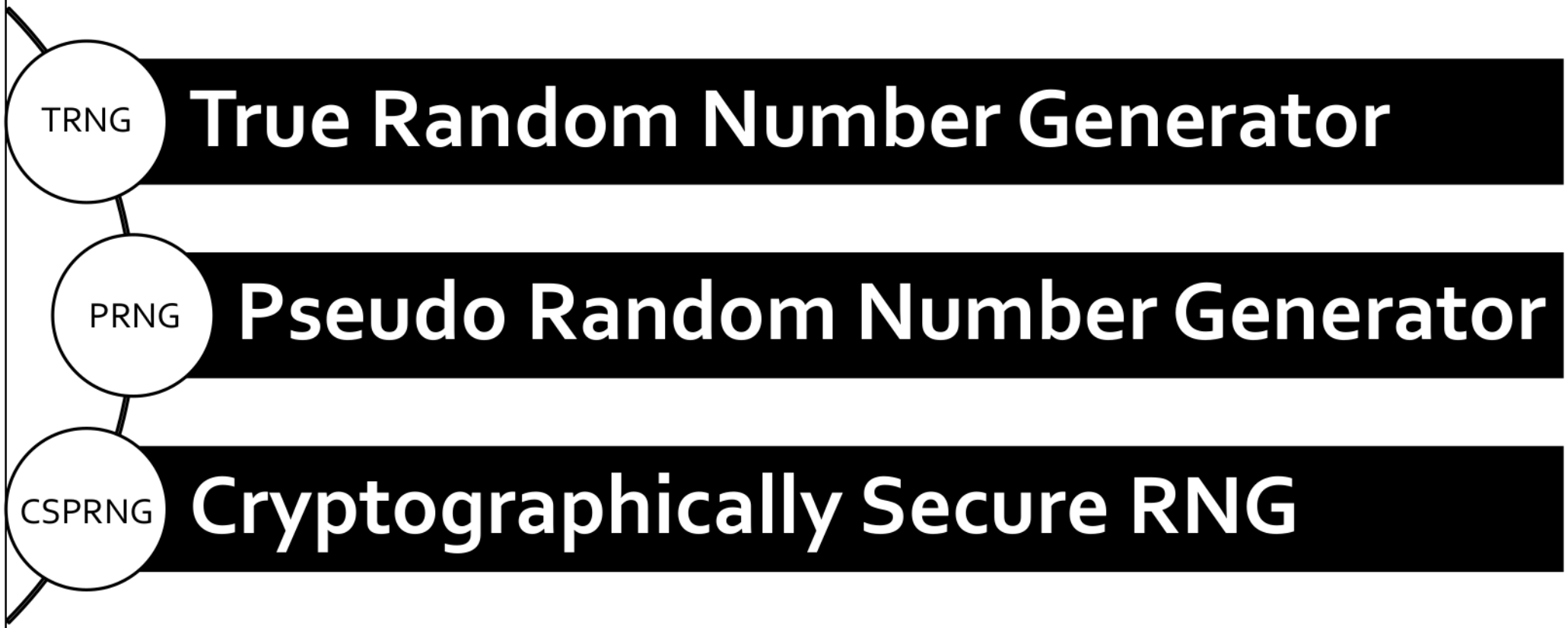
IP-18FKVKRBG

Lecture 5

2022-2023-2

More about PRNG

Random Number Generators (RNGs)



Definition (PRNG)

$$G: \{0,1\}^k \rightarrow \{0,1\}^n$$

- with the following properties:
  - $n >> k$
  - G(x) computationally indistinguishable from true random
  - **the values are uniformly distributed over a defined interval**
  - it is impossible to predict future values based on past or present ones
  - there are no correlations between successive numbers

Probability Distribution

- A **probability distribution** is a mathematical function that provides the probabilities of occurrence of different possible outcomes in an experiment.

- We can pick randomly according to any "probability distribution":

$$P : S \rightarrow R$$

- The probability distribution P assigns a nonnegative probability to each $x \in S$, such that:

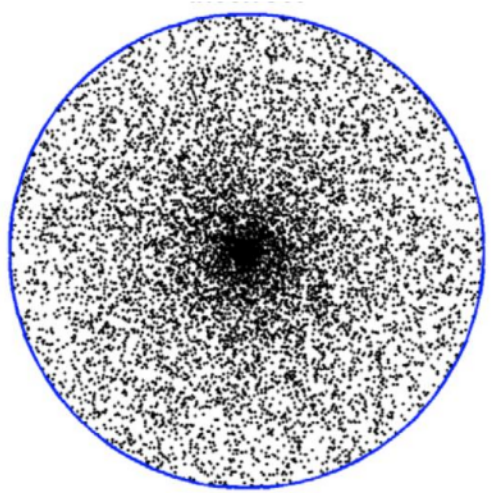
$$\sum_{x \in S} p(x) = 1$$

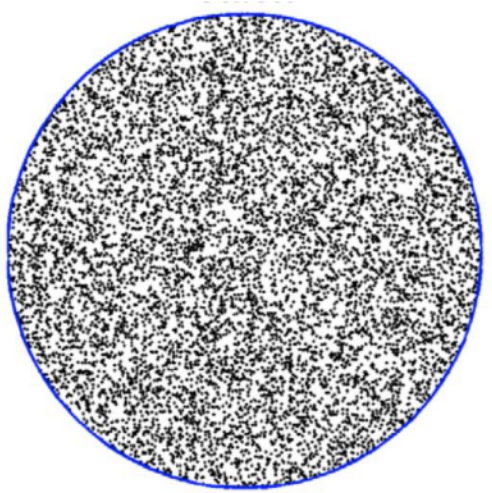
Uniform Distribution

- What if we want to pick something **at random**?

- The uniform distribution U is the probability distribution where everything is picked equally often:

$$If\ |S| = n, then\ \Pr[x = s] = \frac{1}{n}, \qquad \forall s \in_u S$$





Next bit test

A sequence of bits passes the next bit test for at any position i in the sequence, if:

- *an adversary knows the i first bits, and*
- *He cannot predict the* $(i + 1)$*: with probability of success better than* $50\% + \varepsilon$*.*

CSPRNG

Criteria for Cryptographically-Secure Pseudo random number generators (CSPRNG):

- pass statistical randomness tests
- resist against well-known cryptographic attacks
- Every CSPRNG should satisfy the next-bit test.

- ***Yao's theorem:*** *If a generator passing the next-bit test will pass all other polynomial-time statistical tests for randomness.*

Linear Congruential Generators

Linear Congruential Generators

LCG generators are defined by the following recursive formula:

$$\boldsymbol{X_{n+1} \equiv (a\,X_n + c)(mod\ m)}$$

Where:

- a is the multiplier ($0 < a < m$)
- X_0 is a starting seed value
- c is the increment value ($0 \leq c < m$)
- $m > 0$ is the modulus
- It is known that the period of a general LCG is at most m.

Linear Congruential Generators

Linear congruential generators (LCG) are one of the best known pseudorandom number generators. Few example without loss of generality:

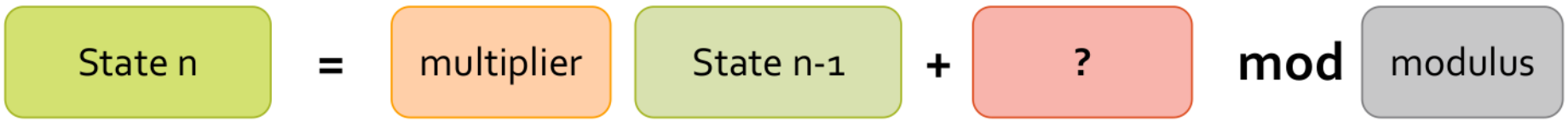
- ANSI C, C++
- Microsoft Visual/Quick C/C++
- Microsoft Visual Basic (6 and earlier)
- Java's java.util.Random
- Turbo Pascal, Borladn Delphi
- Apple CarbonLib
- C++11's minstd rand

Linear Congruential Generators

State n = multiplier State n-1 + increment **mod** modulus

```
def prng_lcg(seed, repeat):
    multiplier = 31706950422767225 7  # the "multiplier"
    increment = 10350850245760656 27  # the "increment"
    modulus = 8512677386048191063  # the "modulus"
    state = seed
    file= open("LCG.rnd","w")

    for _ in range(repeat):
        state = (state * multiplier + increment) % modulus
        file.write(str(state).zfill(19)+"\n")

prng_lcg(300, 300)
```

LCG Cracking: Unknown increment



LCG Cracking: Unknown increment

```
    modulus = 8512677386048191063
    multiplier = 31706950422767225 7
    increment = 10350850245760656 27
    rn = [0, 0, 0, 0, 0, 0]
    file = open("LCG.rnd","r")
    for i in range(6):
        rn[i] = int(file.readline())
```

```
def crack_unknown_increment(states, modulus, multiplier):
    increment = (states[1] - states[0]*multiplier) % modulus
    return increment
```

```
recovered_increment = crack_unknown_increment(rn, modulus, multiplier)
```

LCG Cracking: Unknown multiplier

State n-1 = ? State n-2 + increment **mod** modulus

State n = ? State n-1 + increment **mod** modulus

multiplier = State n - State n-1 / State n-1 - State n-2 **mod** modulus

LCG Cracking: Unknown multiplier

```python
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, x, y = egcd(b % a, a)
        return (g, y - (b // a) * x, x)

def modinv(b, n):  # Modular inverse function
    g, x, _ = egcd(b, n)
    if g == 1:
        return x % n
```

```python
def gcd(x, y): # GCD
   while(y):
       x, y = y, x % y
   return x
```

```python
def crack_unknown_multiplier(states, modulus):
    multiplier = (states[2] - states[1]) * modinv(states[1] - states[0], modulus) % modulus
    increment = crack_unknown_increment(states, modulus, multiplier)
    return increment, multiplier
```

LCG PRNG cracking

```
def crack_unknown_modulus(states):
    diffs = [s1 - s0 for s0, s1 in zip(states, states[1:])]
    zeroes = [t2*t0 - t1*t1 for t0, t1, t2 in zip(diffs, diffs[1:], diffs[2:])]
    modulus = abs(functools.reduce(gcd, zeroes))
    increment, multiplier = crack_unknown_multiplier(states, modulus)
    return modulus, increment, multiplier
```

```
[x] Cracking LCG PRNG
[x] Setting the inputs - random number)
[x] Recovered modulus    :  8512677386048191063
[x] Recovered increment  :  1035085024576065627
[x] Recovered multiplier :  3170695042276722257
```

Mersenne Twister

Mersenne Twister I

- The Mersenne Twister is a pseudorandom number generator (PRNG)
- It has a very long period $2^{19937} - 1$
- **k-distributed** to 32-bit accuracy for every $1 \leq k \leq 623$
- Really fast
- Python random.Random() is based on MT.

Mersenne Twister II

- Excellent statistical properties (indistinguishable from truly random)
- However, it is NOT CPRNG.

Steps to Crack MT:

a) **randcrack** package to crack it in python!

b) feed cracker with 624 randomly generated numbers (32-bit integers)

c) After submitting 624 integers it will be ready for predicting new numbers.

Mersenne Twister Cracker

```
from randcrack import RandCrack

random.seed(100)
rc = RandCrack()

print("[x] feeding the cracker with 624 randomly generated numbers")
for i in range(624):
    r = random.getrandbits(32)
    rc.submit(r)

print("[x] Now, it is ready to predict new randomly generated numbers ..")
print("\n[x] Random Number: {}\n[x] Predicted number: {}"
    .format(random.randrange(0, 4294967295), rc.predict_randrange(0, 4294967295)))
```

ELTE
EÖTVÖS LORÁND
UNIVERSITY

CRYPTOGRAPHY AND SECURITY

Practice

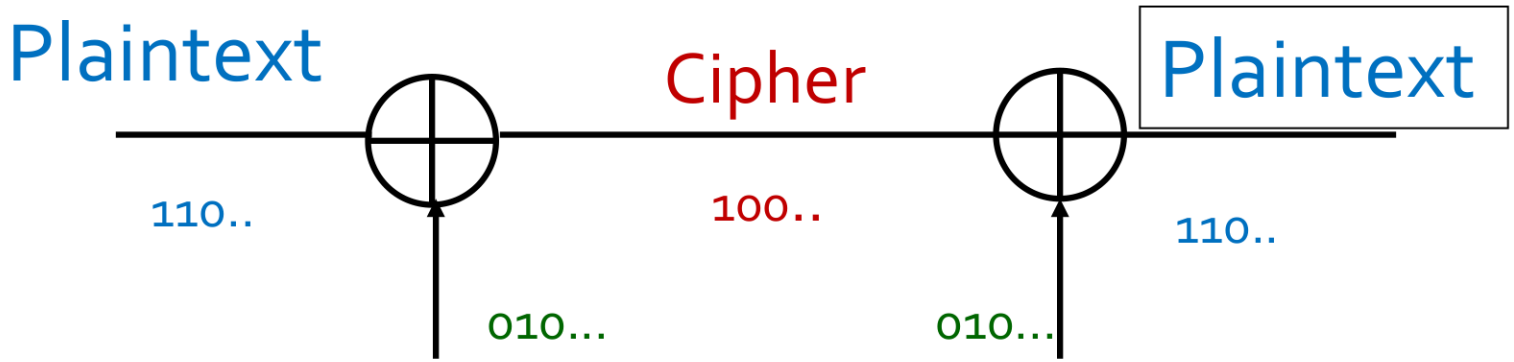
IP-18FKVKRBG

Lecture 6

2022-2023-2

Stream Cipher

OTP: One Time Pad

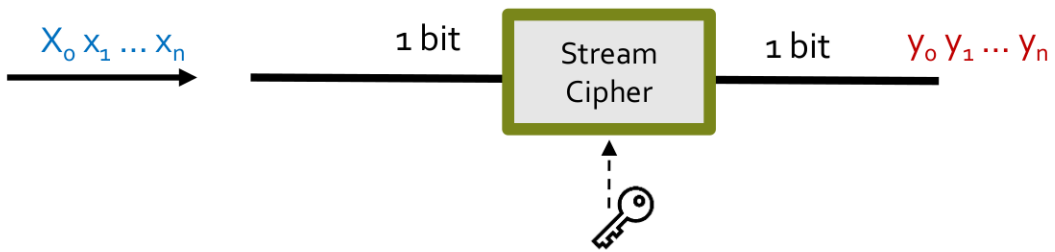


- Provable secure provided
  - Key K is random
  - Key K is as long as the message
  - Key K is used only one time

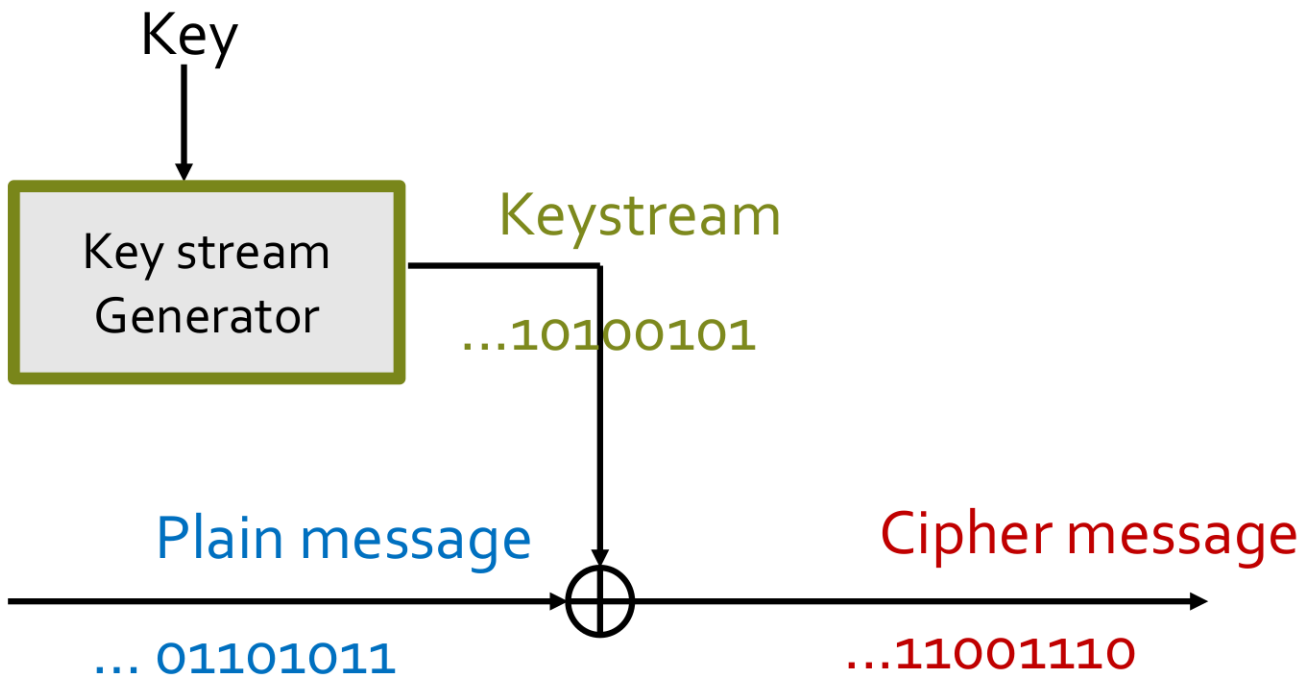
K = 010...

Stream ciphers

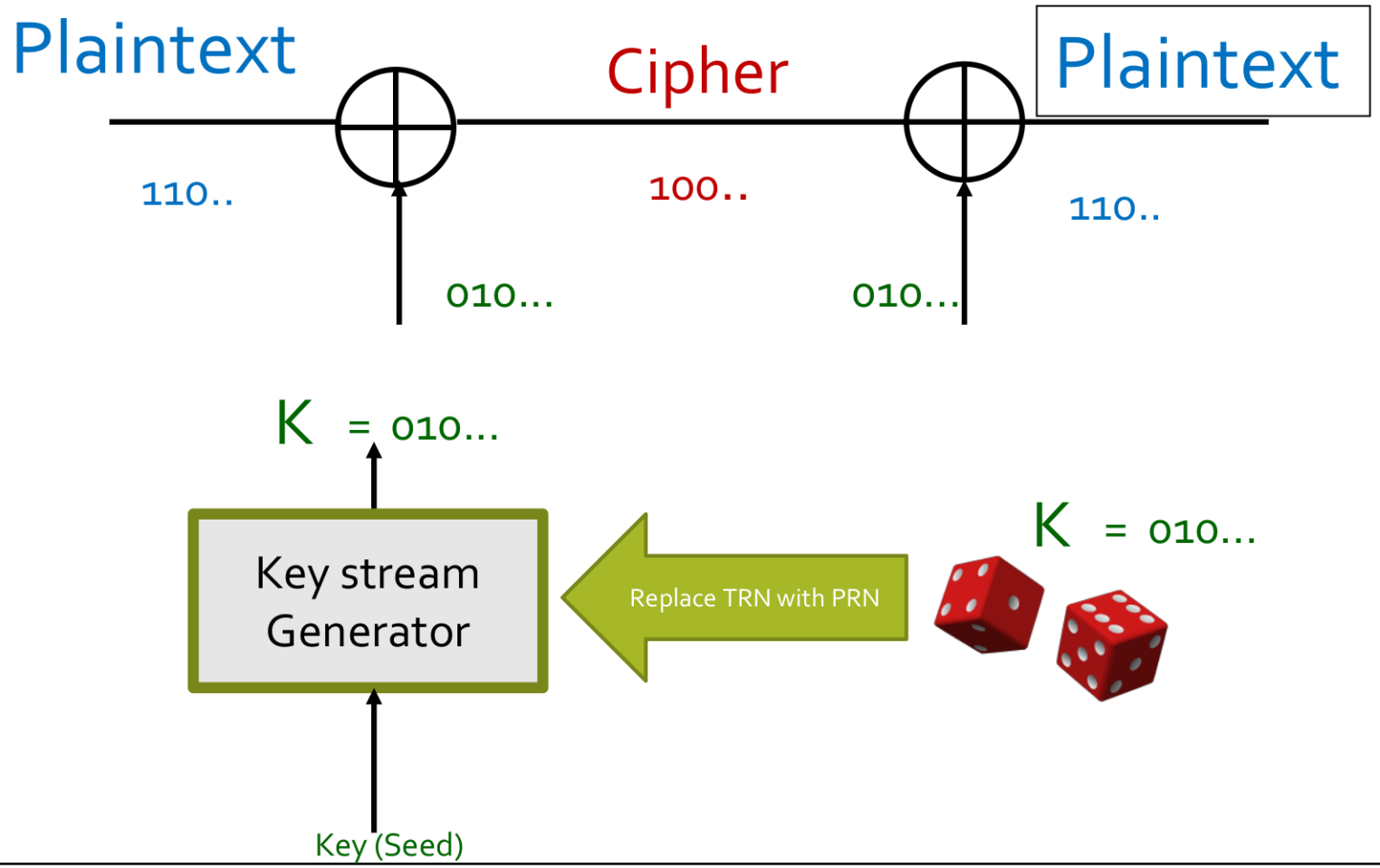
- In One-Time Pad, a key is a random string of length at least the same as the message
- Stream ciphers:
  - Idea: replace "rand" by "pseudo rand"
  - Use a "Pseudo" Random Number Generator
  - G: $\{0,1\}^k \rightarrow \{0,1\}^n$
    - expand a short (e.g., 128-bit) random seed 'k' into a long (e.g., 10^6 bit) string that "looks random"
  - Secret key is the seed



Key Generator



Stream Cipher

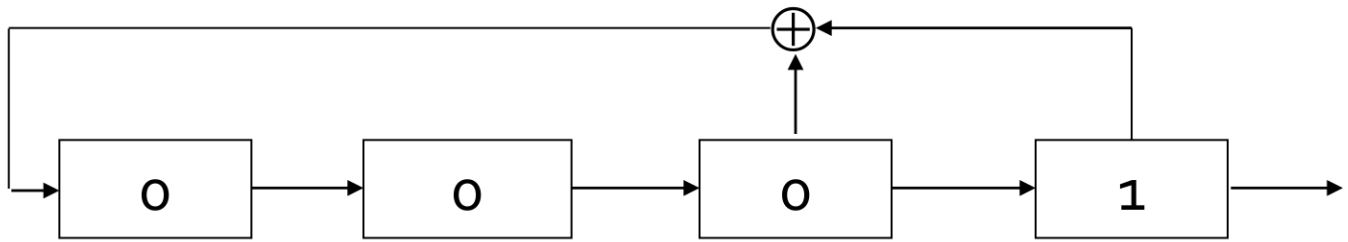


LFSR Introduction

LFSR (Linear Feedback Shift Register)

- A linear-feedback shift register (LFSR) is a shift register whose input bit is a linear function of its previous state.
- The most commonly used linear function of single bits is exclusive-or (XOR).
- An LFSR with a well-chosen feedback function can produce a sequence of bits that appears random and has a **very long cycle**.
- The **Berlekamp–Massey** algorithm is an algorithm that will find the shortest linear feedback shift register (LFSR) for a given binary output sequence.

LFSR Example



- The seed is the key

- Starting with 1000, the output stream is:
- 1000 1001 1010 1111 000
- Repeat every $2^4 - 1$ bit

Warming up code

```
def shift_and_print(iv):
    while iv:
        ot = 1 & iv        # save the rightmost bit
        iv >>= 1           # shift
        print(ot)
```

```
def shift_and_yield(iv):
    while iv:
        ot = 1 & iv        # save the rightmost bit
        iv >>= 1           # shift
        yield ot
```

```
shift_and_print(int(hex(0x63), 16))
shift_and_print(int('1001110', 2))

print('Output in one line = ',

      ''.join(str(output) for output in  shift_and_yield(int('1101001', 2))))
```

LFSR

```python
def lfsr_generate(seed, mask, n):
    seed_int = int(seed, 2)
    mask_int = int(mask, 2)
    nbits = len(seed)

    state = seed_int
    while n > 0:
       output = 1 & state # get the most right bit

       _mask, _state, new_bit = mask_int, state, 0
       while _mask:
           new_bit ^= (1 & _mask) * (1 & _state)
           _mask >>= 1
           _state >>= 1

       state = state >> 1 | new_bit << (nbits - 1)

       yield output, state
       n -= 1
```

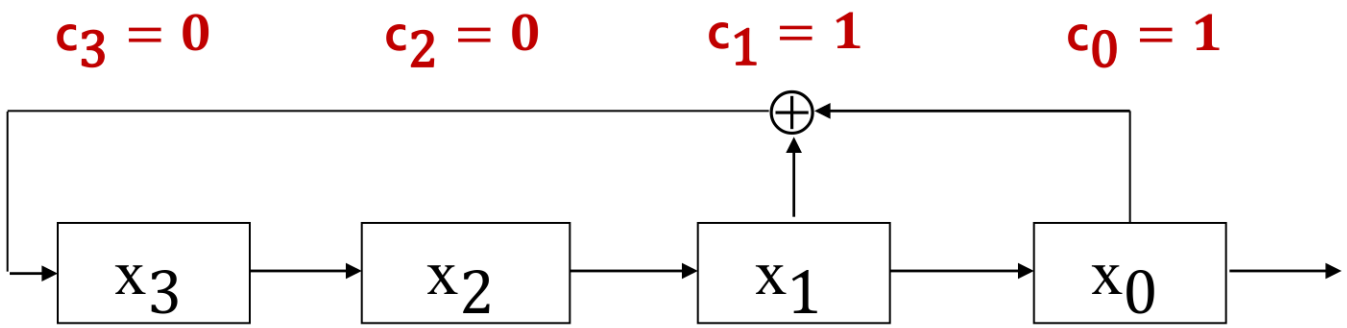
```python
seed = '0001'
mask = '0101'
samples = 20

key = lfsr_generate(seed, mask, samples)
key_str = ''.join(str(x) for x, _ in key)
key_hex = hex(int(key_str, 2))[2:]
```

LFSR Cracking

LFSR: Cryptanalysis I

- A LFSR can be described as:

$$x_{n+i} = \sum_{j=0}^{m-1} c_j x_{i+j} \bmod 2$$



$$x_4 = c_0x_0 + c_1x_1 + c_2x_2 + c_3x_3 \bmod 2$$

$$x_4 = x_0 + x_1 \bmod 2$$

LFSR: Cryptanalysis II

- Vulnerable to known-plaintext attack

- Knowing **2n output** bits, one can
- construct **n linear equations** with **n unknown variables** $c_0, \ldots, c_{n-1}$
- recover $c_0, \ldots, c_{n-1}$

LFSR: Cryptanalysis III

- Can determine the minimum polynomial $f(x)=x^n+c_{n-1}x^{n-1}+\ldots+c_o$ of a sequence (s_t) from 2n successive bits $\mathbf{s_0, s_1, \ldots, s_{2n-1}}$

$$\begin{bmatrix} s_0, & s_1, \ldots, s_{n-1} \\ s_1, & s_2, \ldots, s_n \\ \ldots\ldots\ldots\ldots\ldots & \\ s_{n-1}, & s_n, \ldots, s_{2n-2} \end{bmatrix} \begin{bmatrix} c_o \\ c_1 \\ \ldots \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} s_n \\ s_{n+1} \\ \ldots \\ s_{2n-1} \end{bmatrix}$$

- Matrix has rank **n** if minimum polynomial has rank n
- There exists a very efficient algorithm due to **Berlekamp-Massey** to calculate $c_o, c_1, \ldots, c_{n-1}$ in $O(n^2)$ operations

LFSR: Cryptanalysis IV

```
from berlekampmassey import bm


poly = list(bm('10001010001010001010'))[-1]
print("[*] Recovering secret key  : ",poly[::-1])
```

ELTE
EÖTVÖS LORÁND
UNIVERSITY

CRYPTOGRAPHY AND SECURITY

Practice

IP-18FKVKRBG

Lecture 7

2022-2023-2

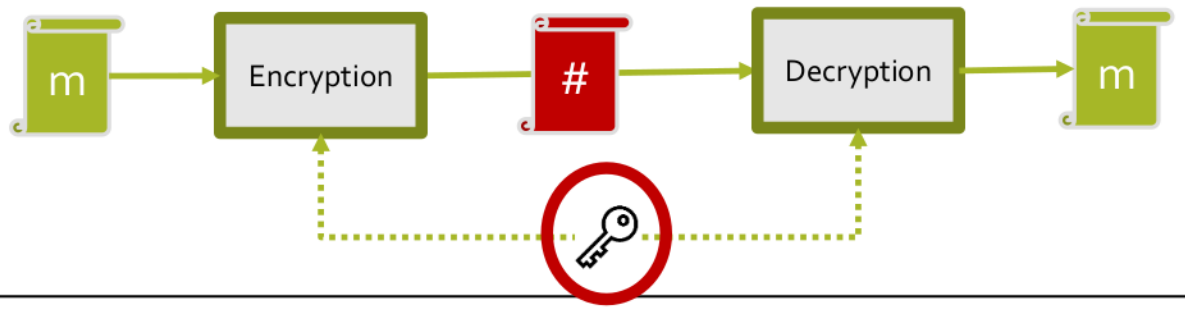
Symmetric Cryptography

Symmetric Cryptography

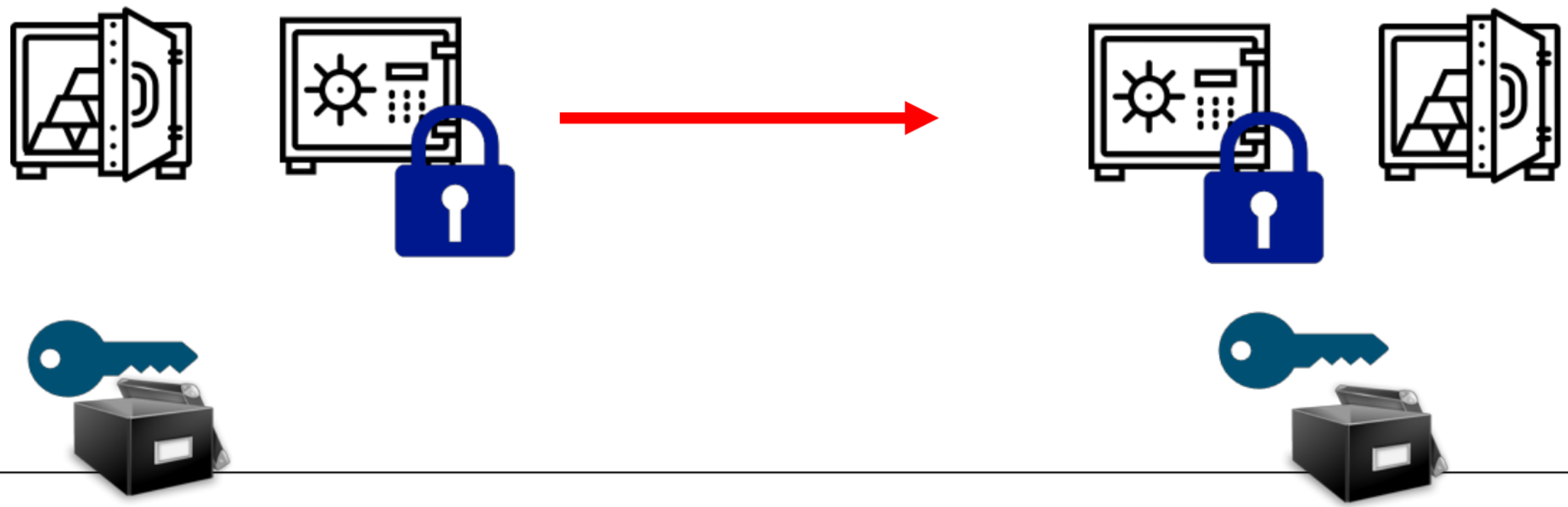
- **Symmetric-key encryption** is a type of encryption where only one key (a secret key) is used to both encrypt and decrypt electronic information.

- Anyone who knows the secret key can decrypt the secret message.

- With symmetric-key encryption, the encryption key can be calculated from the decryption key, and vice versa.

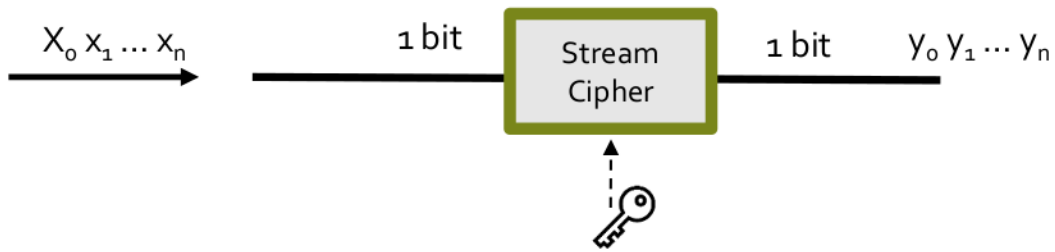


Symmetric Cryptography I

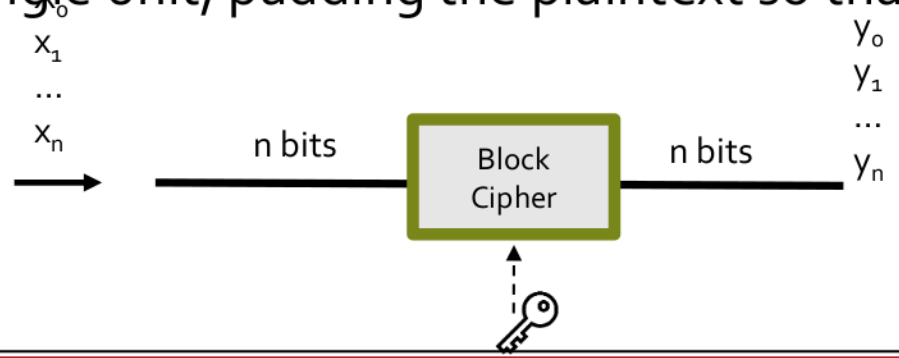


Symmetric Cryptography II

- Stream Cipher: Encrypts one bit at time



- Block Cipher: Encrypts a set of bits (i.e. a block of data, e.g. 64 or 128 bits) at time, encrypt them as a single unit, padding the plaintext so that it is a multiple of the block size.

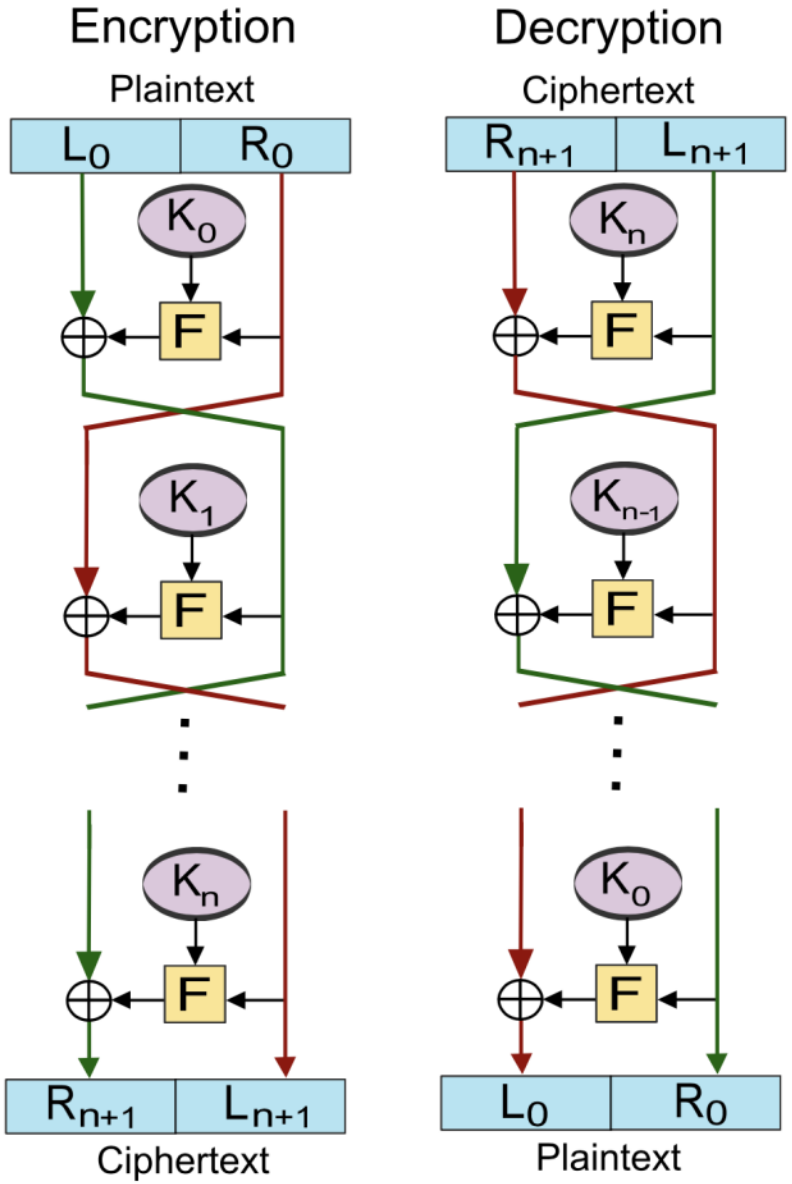


Data Encryption System (DES)

Feistel-network

The entire operation is guaranteed to be invertible.

that is, encrypted data can be decrypted, **even if the round function is not itself invertible**.



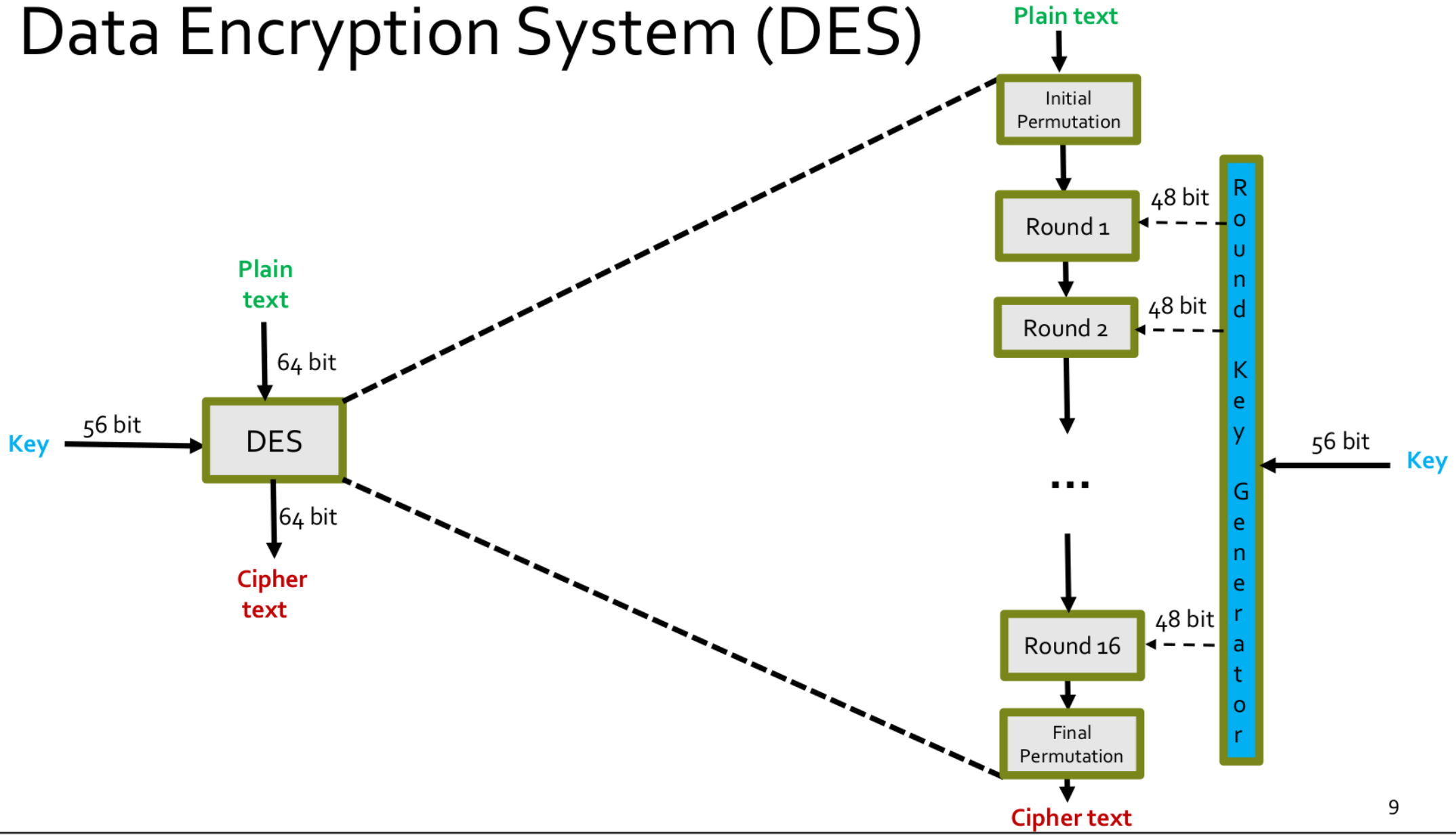
Data Encryption System (DES)

- based on an earlier design by Horst Feistel.
- In 1976, DES is approved as a standard, 1rst published in 1977

- Encrypts **blocks** of size **64 bits**.
- Uses a **key** of size **56 bits**.
- The same (56-bits) key is used both for encryption and decryption.
- Includes 16 rounds. Each of them performs the same set of operations.
- A 48-bits subkey is generated for each round. (Derived from the main key)

- Today DES considered as an insecure algorithm

Data Encryption System (DES)



Some of the Attacks on DES

- In 1977, Diffie and Hellman proposed a machine costing an estimated US$20 million which could find a DES key in a single day.
- In 1991, Biham and Shamir proposed differential cryptanalysis attack that required 247 chosen ciphertexts.
- In 1997, distributed.net could break DES key in 3 months (Costs 250k$)
- In 1998, Deep Crack breaks a DES key in 56 hours
- In 1999, Deep Crack along with distributed.net could break DES key in 22 hours and 15 minutes
- In 2006, COPACOBANA could break DES key in 7 days (Costs 10k$)
- In 2016, using hashcat could recover a key in an average of under 2 days

Alternatives to DES

Algorithm	Input/ Output length	Notes
AES (Rijndael)	128	Standard replacement to DES
Mars	128	AES Finalist
RC6	128	
Serpent	128	
Twofish	128	

Algorithm	Input/ Output length
Triple DES	64
IDEA	64

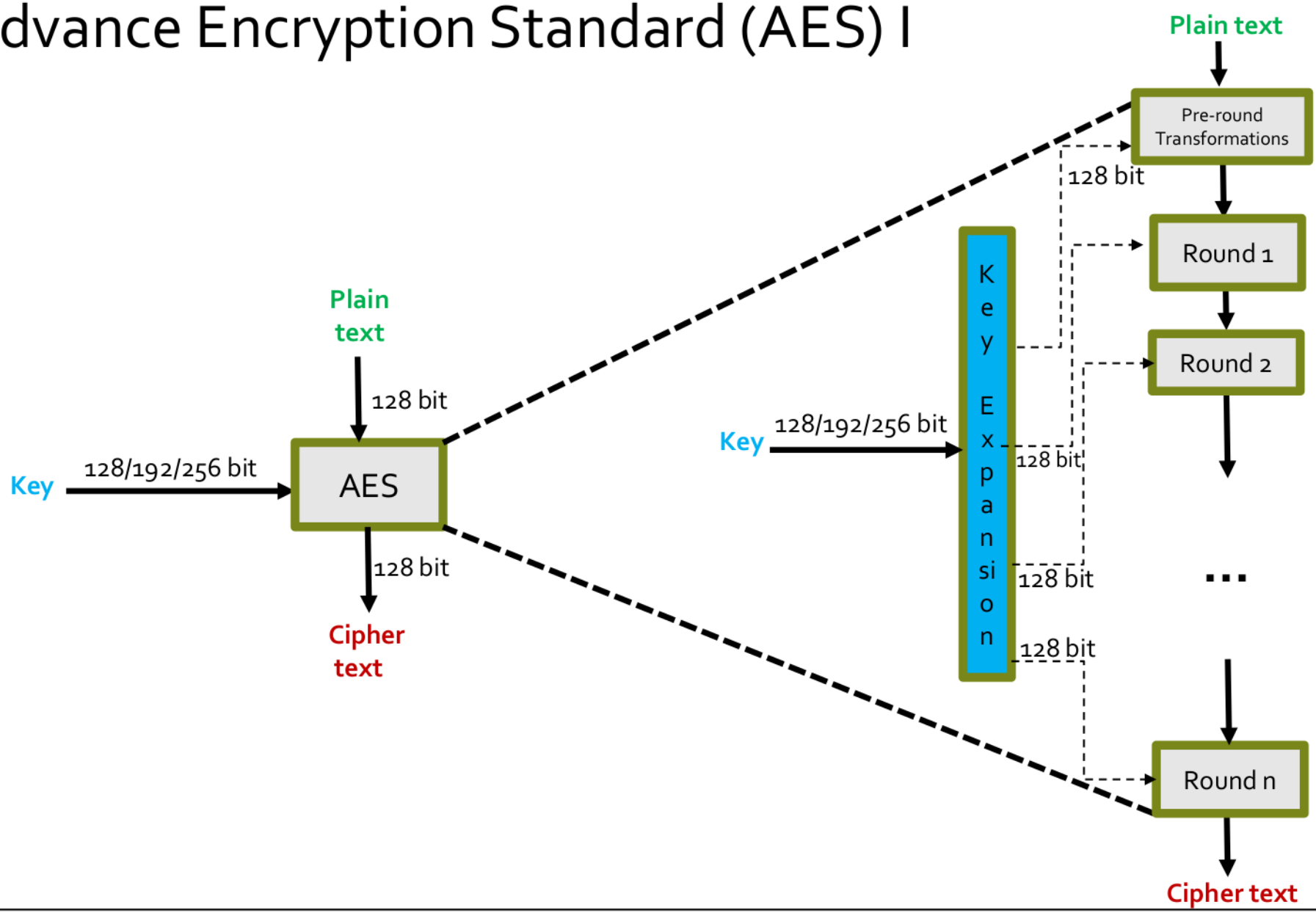
Advance Encryption Standard (AES)

Advance Encryption Standard (AES)

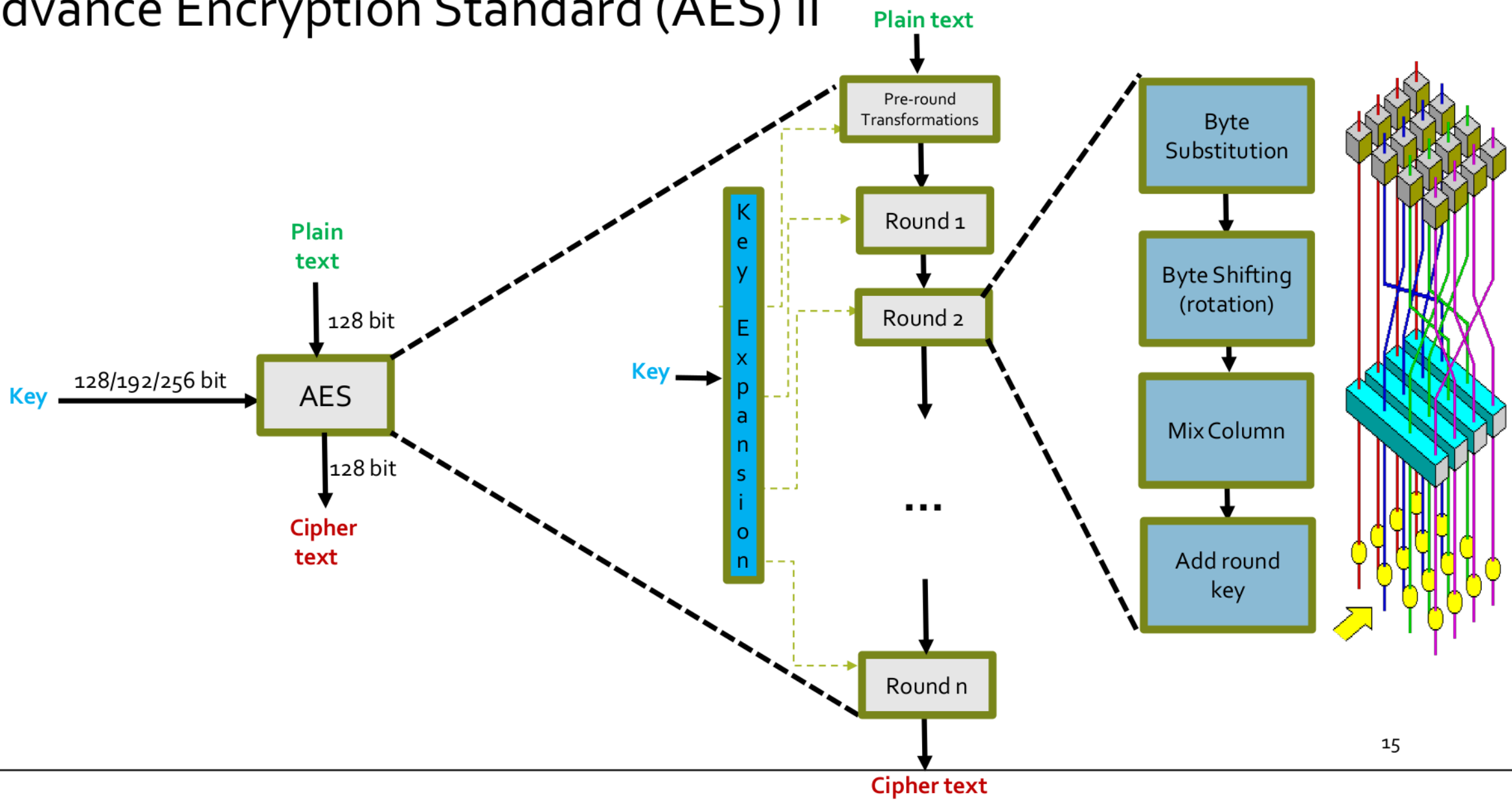
- **Rijndael** developed by Vincent Rijmen and Joan Daemen
- First published in 1998
- The winner among the 5 AES finalists announced in 1999
- Became effective in 2002
- The number of rounds (n) depends on the length of the key:

Key length (in bits)	Number of rounds (n)
128	10
192	12
256	14

Advance Encryption Standard (AES) I



Advance Encryption Standard (AES) II



Block cipher modes (for DES, AES, …)

- Confidentiality only modes:
  - ECB mode: Electronic Code Book mode
  - CBC mode: Cipher Block Chaining mode
  - CFB mode: Cipher FeedBack mode
  - OFB mode: Output FeedBack mode
  - CTR mode: Counter mode
- Authenticated encryption with additional data (AEAD) modes:
  - Galois/counter mode (GCM)
  - Counter with cipher block chaining message authentication code (CCM)

Cipher Block Chaining (CBC)

- The plaintext is broken into blocks, based on the block size of the used cipher algorithm.
- Each block is XORed (chained) with the ciphertext of the previous block before encryption:

$$C_i = E_K(C_{i-1} \oplus P_i)$$

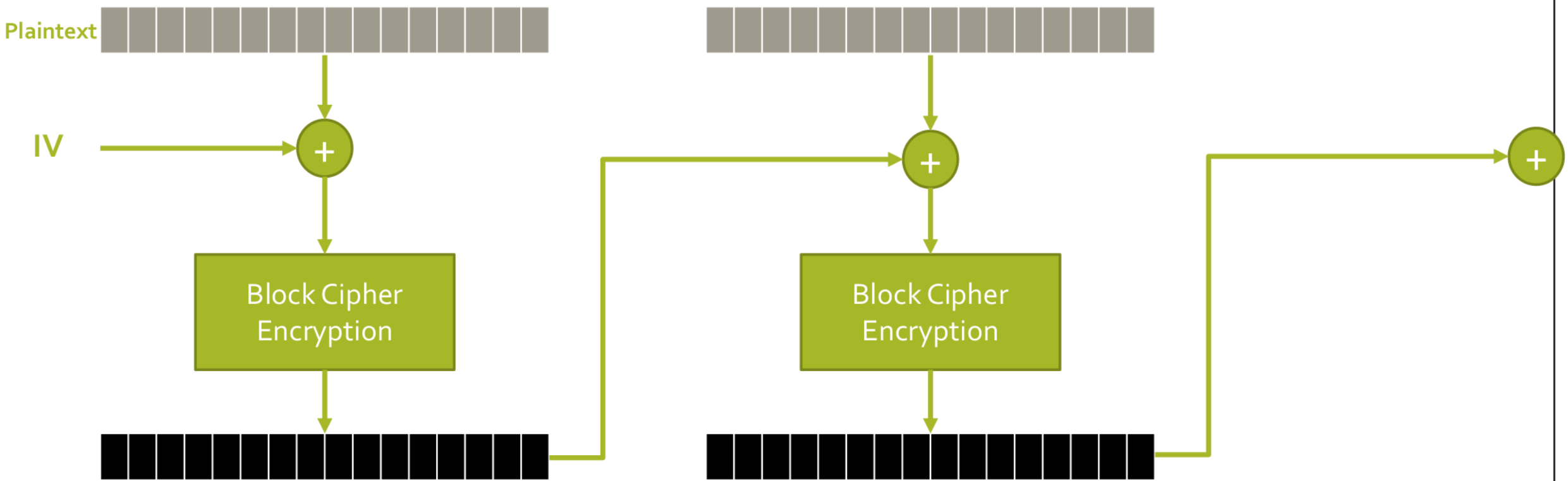
- IV is used to start initiate the process

$$C_1 = E_K(IV \oplus P_1)$$

- Decryption is done as:

$$P_i = C_{i-1} \oplus D_K(C_i)$$

CBC Mode of Encryption



AES (CBC Mode)

Padding

```
import secrets
from Crypto.Cipher import AES

BLOCK_SIZE = 16  # Bytes
pad = lambda s: s + (BLOCK_SIZE - len(s) % BLOCK_SIZE) * \
                chr(BLOCK_SIZE - len(s) % BLOCK_SIZE)
unpad = lambda s: s[:-ord(s[len(s) - 1:])]
```

Encryption

```
    iv =  secrets.token_bytes(16)
    Enc = AES.new(key, AES.MODE_CBC, iv)
    data = pad(plaintext).encode()
    ciphertext = Enc.encrypt(data)
    ciphertext_hex = iv.hex() + ciphertext.hex()
```

Decryption

```
    iv = bytes.fromhex(ciphertext[:32])
    ciphertext = bytes.fromhex(ciphertext[32:])
    Dec = AES.new(key, AES.MODE_CBC, iv)
    pt = Dec.decrypt(ciphertext)
```

ELTE
EÖTVÖS LORÁND
UNIVERSITY

CRYPTOGRAPHY AND SECURITY

Practice

IP-18FKVKRBG

Lecture 8

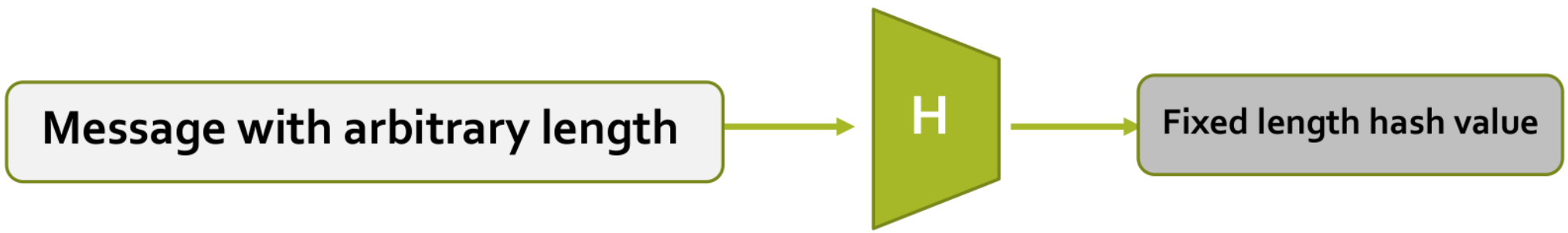
2022-2023-2

Hash Functions

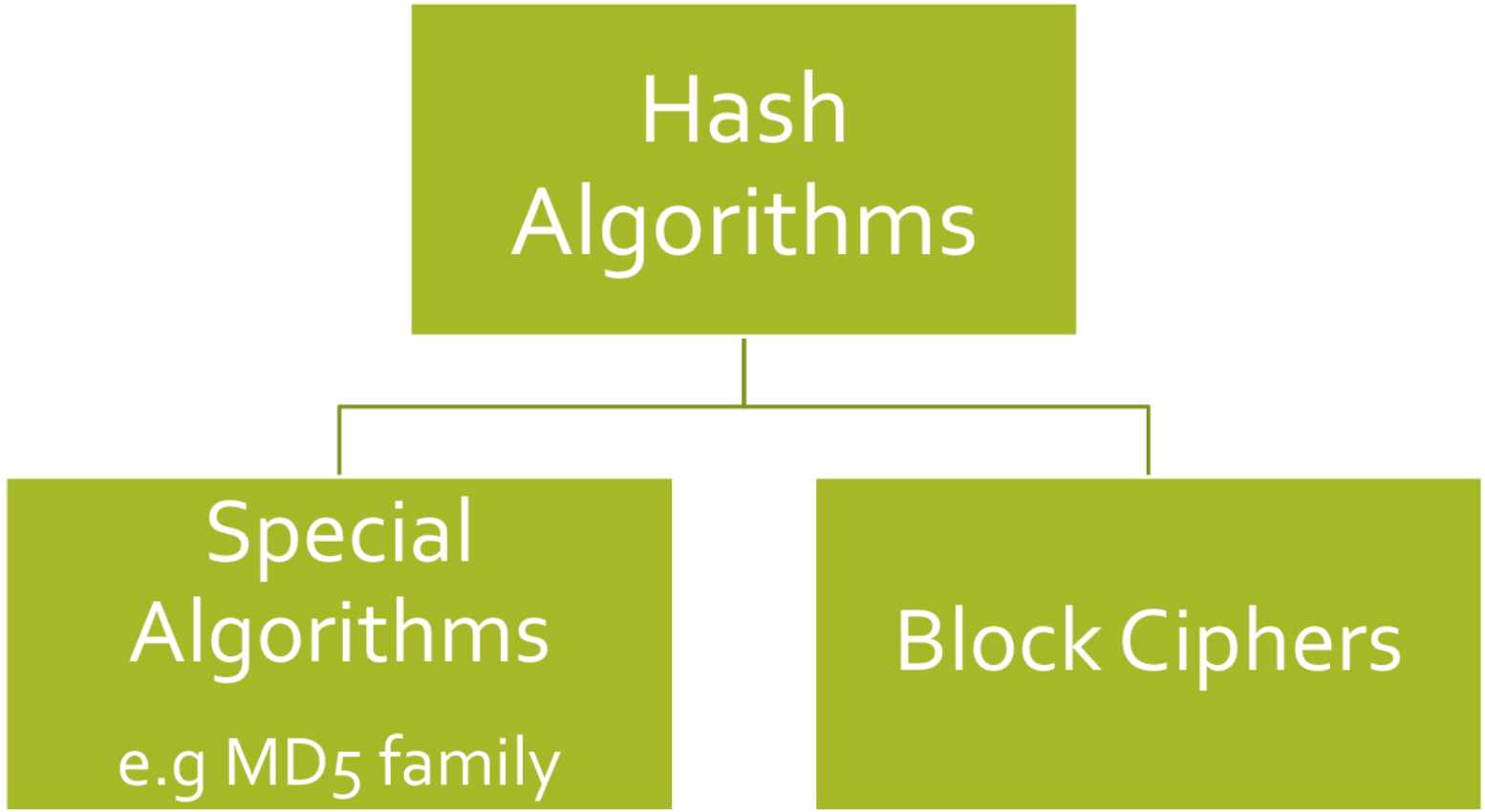
Hash Function

- A **hash function** is a computationally efficient function mapping binary strings of arbitrary length to binary strings of some fixed length, called *hash-values*.

$$\boldsymbol{H} : \{0,1\}^* \rightarrow \{0,1\}^k$$

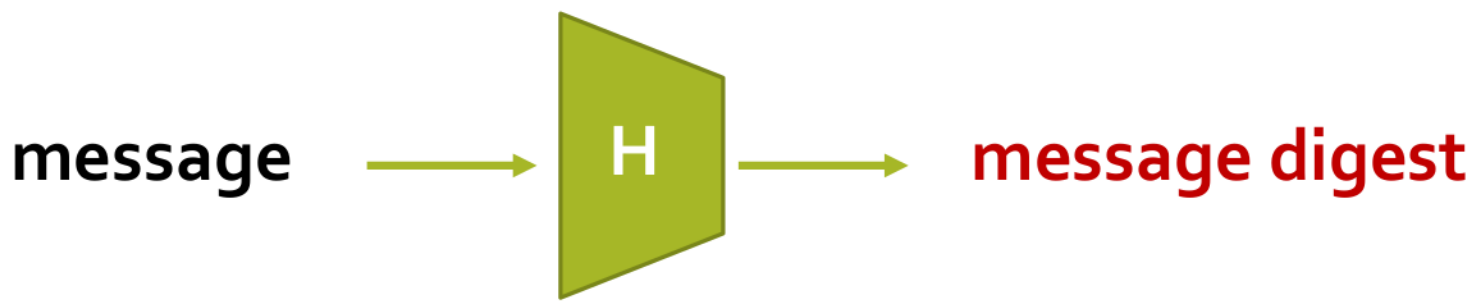


Hash Function Algorithms



Cryptographic hash function

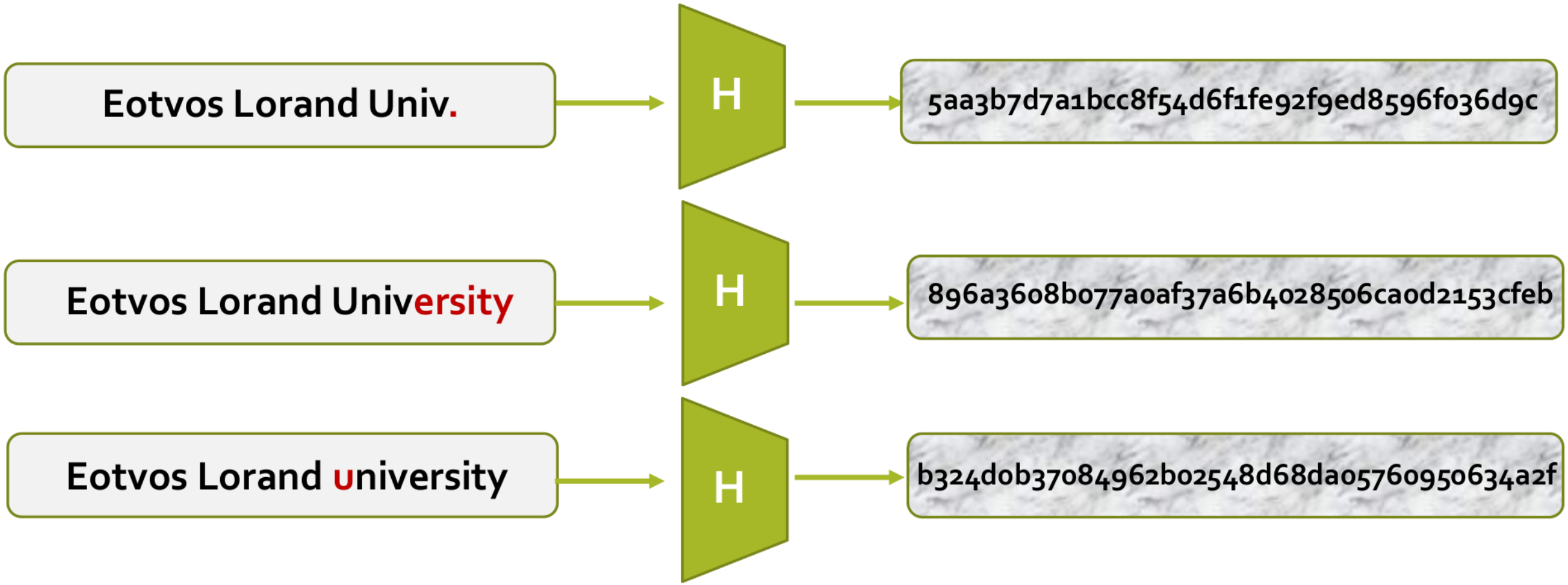
- A cryptographic hash function is a mathematical algorithm that maps data of arbitrary size to a bit string of a fixed size (a hash function) which is designed to also be **one-way function**, that is, a function which is **infeasible to invert**.
- The input data is often called the **message**, and the output (the hash value or hash) is often called the **message digest** or simply the digest.



Properties of cryptographic hash functions

- it is quick to compute the hash value for any given message.
- it is infeasible to generate a message from its hash value except by trying all possible messages.
- a small change to a message should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value.
- it is infeasible to find two different messages with the same hash value.

Principal behavior of hash functions



* sha1

Applications of one-way hash

- Password files (one way)
- Data integrity
- Digital signatures (collision resistant)
- Keyed hash for message authentication

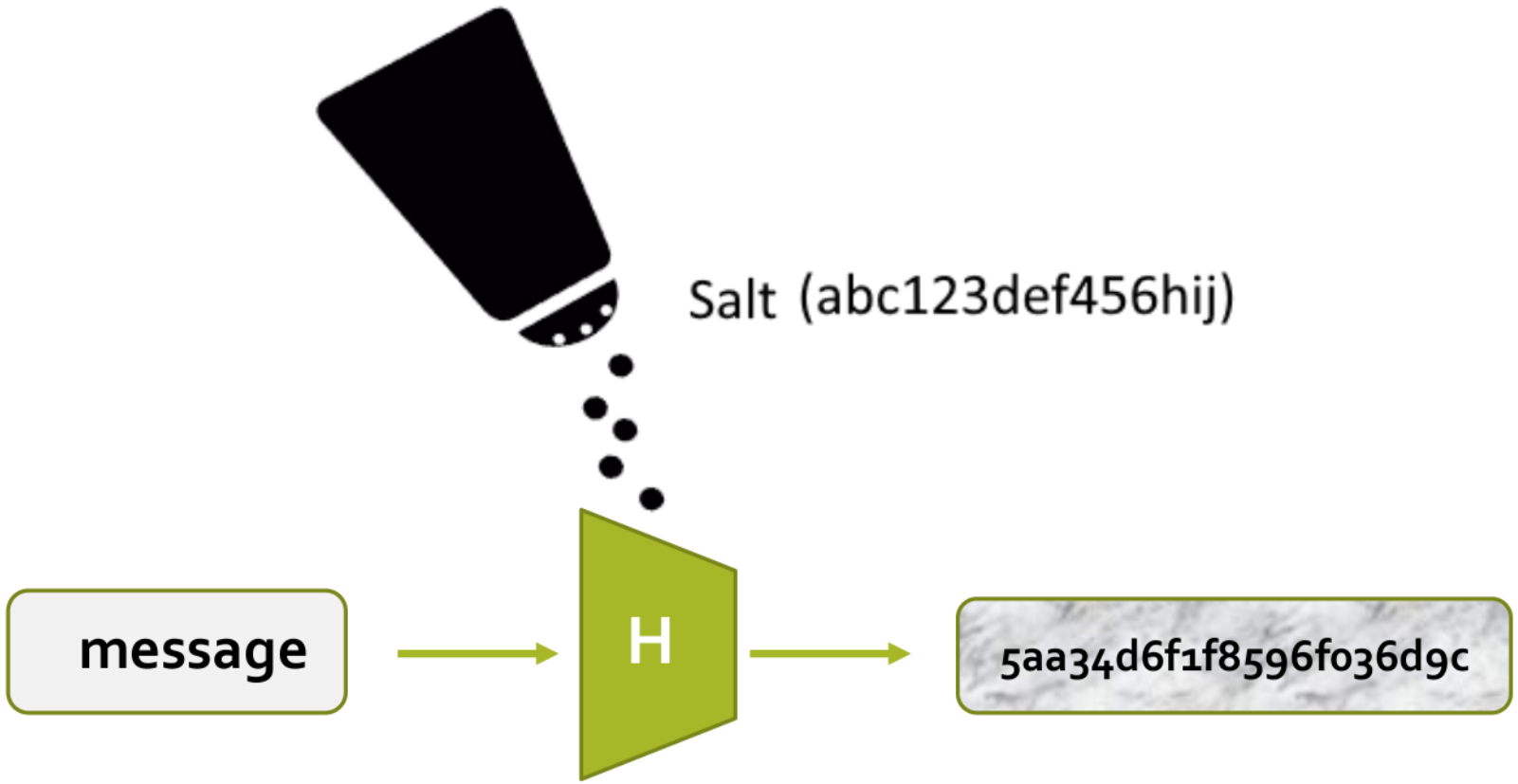
A simple Hashing

```
from hashlib import sha1


def generate_hash(name):
   return sha1(name.encode()).hexdigest()



def test():
    name = input('Enter your name: ')
    print('The hashed value of your name is = ', generate_hash(name))

test()
```

Simple Password Hashing

```
from hashlib import sha1

def hash_password(password):
   return sha1(password.encode()).hexdigest()

def check_password(stored_hashed_password, plain_password):
   password_validity = stored_hashed_password == sha1(plain_password.encode()).hexdigest()
   return password_validity

def test(entered_password):
    hashed_password = 'd20a09545c7aff14a4f596ddba19296d58f6c101'
    valid_pass = check_password(hashed_password, entered_password)
    if valid_pass:
      print('[x] Password is valid')
    else:
      print('[x] Password is not valid')

test('secret password')
```

Salted Hashing



Salted Hashing

```
import uuid
from hashlib import sha1

def hash_salted_password(password):
    salt = uuid.uuid4().hex  # generate a random unique ID as a 'salt' value
    return sha1(salt.encode() + password.encode()).hexdigest() + ':' + salt

def check_salted_password(stored_hashed_password, plain_password):
    password_hash, salt = stored_hashed_password.split(':')
    password_validity = password_hash == sha1(salt.encode() + plain_password.encode()).hexdigest()
    return password_validity

def generate_salted_password(plain_password):
    print('[x] The plain password : ', plain_password)
    hashed_password = hash_salted_password(plain_password)
    print('[x] Stored in the db : ', hashed_password)

generate_salted_password('secret password')
```

ELTE
EÖTVÖS LORÁND
UNIVERSITY

CRYPTOGRAPHY AND SECURITY

Practice

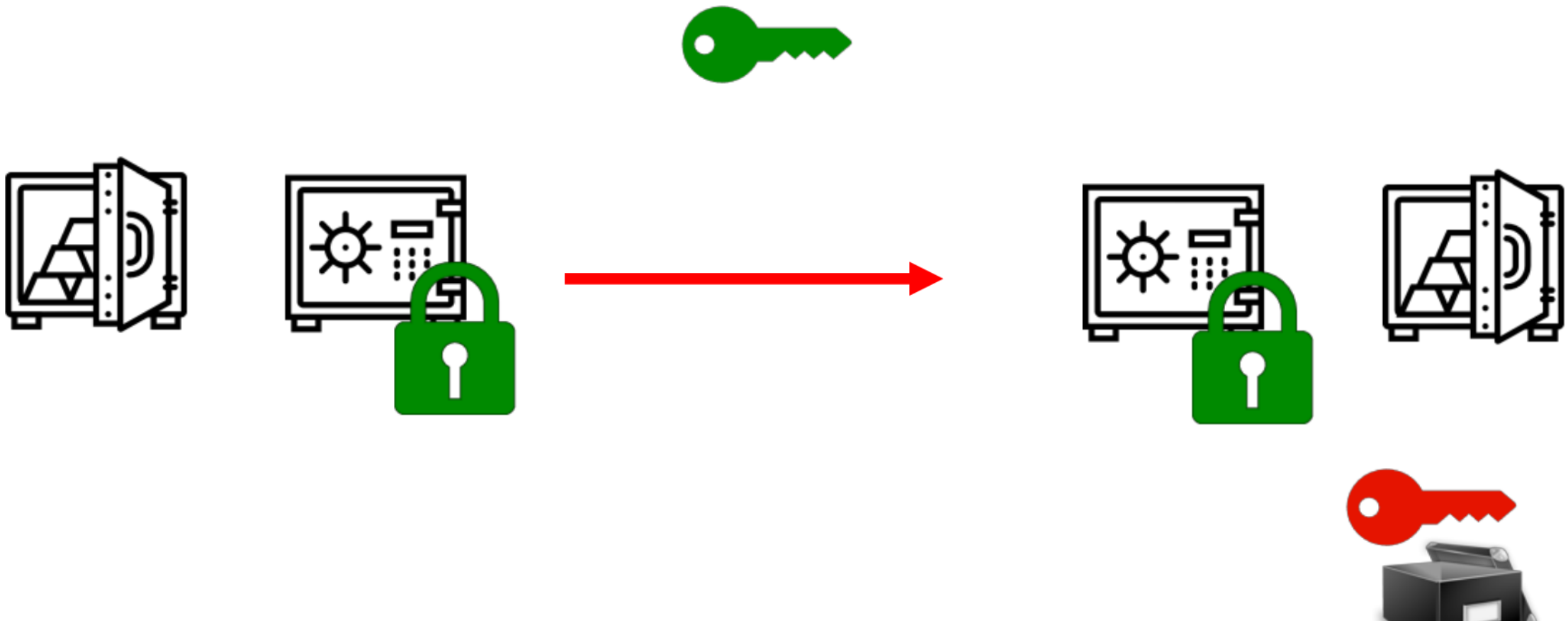
IP-18FKVKRBG

Lecture 9

2022-2023-2

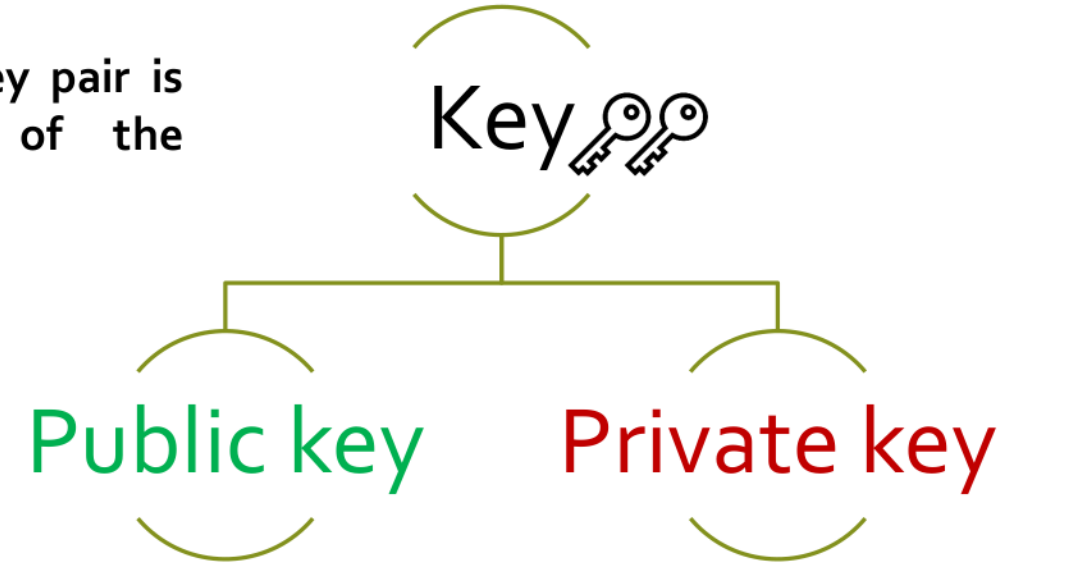
Asymmetric Cryptography

Asymmetric Ciphers



Asymmetric (Public-Key) Cryptography

In key generation, a key pair is generated for each of the members of the system.

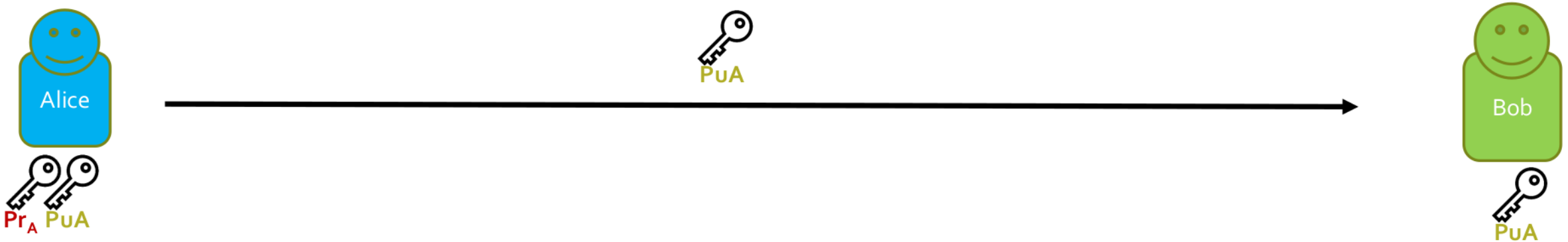


(Encrypt) ⟶ (Decrypt)

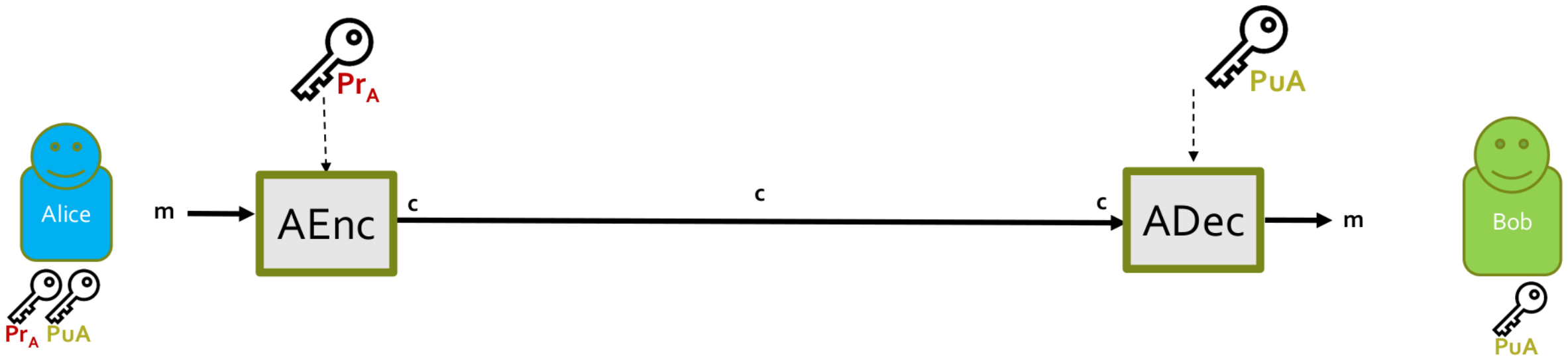
(Decrypt) ⟵ (Encrypt)

A text Can NOT be encrypted and then decrypted by the SAME key

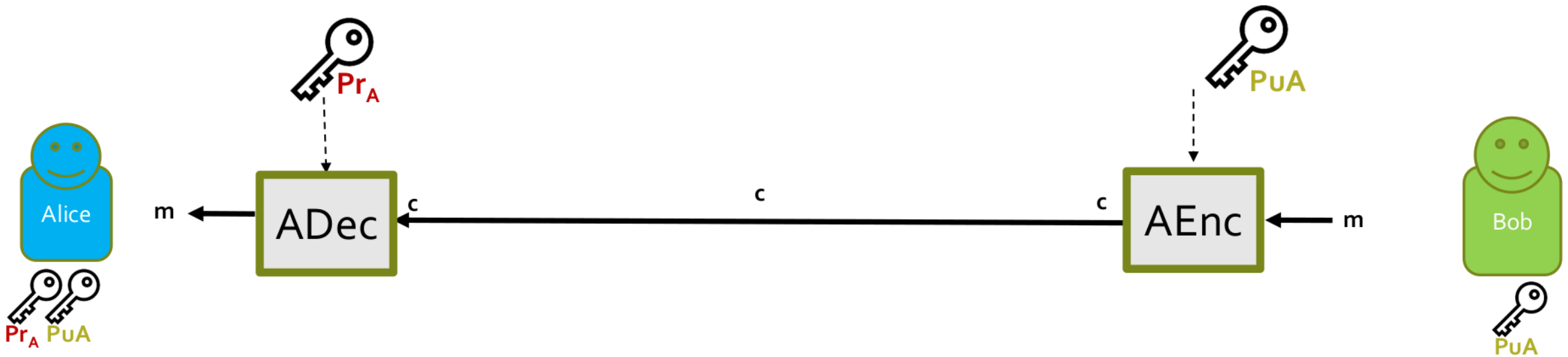
Public-Key Encryption: Sharing public key



Public-Key Encryption: Sender Authentication



Public-Key Encryption: Confidentiality



RSA

RSA: Key generation

- Choose two unequal large prime numbers p and q
- Compute modulo $n = p.q$
- Compute totient function $\varphi(n) = \varphi(p.q) = (p-1)(q-1)$

- Generate numbers e, d such that
  - $1 < e < \varphi(n)$ and e relatively prime to $\varphi(n)$
  - Find d such that $ed \equiv 1 \pmod{\varphi(n)}$ → it only exist if gcd(e, $\varphi(n)$) = 1
  - $x^{ed} \equiv x \bmod n$

RSA: "schoolbook" Encryption and Decryption

- Public encryption key $\langle n, e \rangle$
  - Encrypt($\langle n, e \rangle$, x) = $x^e \bmod n$

- Private decryption key $\langle n, d \rangle$
  - Decrypt($\langle n, d \rangle$, y) = $y^d \bmod n = x^{ed} \bmod n = x$

RSA: Example

Preparation (Bob Side)

- Bob chooses p = 41, q = 61, → and gets n = 2501, $\varphi(n) = 2400$
- He chooses e = 23, → and gets d = 2087
  - By choosing other values of e we get other values of d.
- Alice receives n and e, from Bob

Alice (Sender) Side

- Plaintext = **100**
- Encryption: $100^{23} \bmod 2501 =$ **2306**

Bob (Receiver) Side

- Decryption: $2306^{2087} \bmod 2501 =$ **100**

RSA Coding: Necessary functions

```
from sympy import randprime
from math import gcd

def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, x, y = egcd(b % a, a)
        return (g, y - (b // a) * x, x)

def modinv(b, n):
    g, x, _ = egcd(b, n)
    if g == 1:
        return x % n
```

RSA Coding: Generate two prime numbers

It makes sure that:

- both generated prime numbers are not the same, and
- their product does not exceed the final number of bits (key size)

```
    while prime_1 == prime_2 or (prime_1 * prime_2) > 2**key_size:
        prime_1 = randprime(3, 2**key_size/2)
        prime_2 = randprime(3, 2**key_size/2)
```

Note: For in lecture testing choose a small key size.

RSA Coding: Generating the public parameter

```
def generate_public_exponents(totient):
    public_exponent = 0
    for e in range(3, totient-1):
        if gcd(e, totient) == 1:
            public_exponent = e
            break
    return public_exponent
```

RSA Coding: Generating the e and d parameters

```
public_exponent = generate_public_exponents(totient)

private_exponent = modinv(public_exponent, totient)
```

RSA Coding: Encryption and decryption

```
def rsa_encrypt(plain_text, rsa_modulus, public_exponent):
    cipher = ''.join(chr((ord(ch)**public_exponent) % rsa_modulus)
            for ch in plain_text)
    return cipher.encode().hex()

def rsa_decrypt(cipher_text, rsa_modulus, private_exponent):
    return ''.join(chr((ord(ch)**private_exponent) % rsa_modulus)
                   for ch in bytearray.fromhex(cipher_text).decode())
```

ELTE
EÖTVÖS LORÁND
UNIVERSITY

CRYPTOGRAPHY AND SECURITY

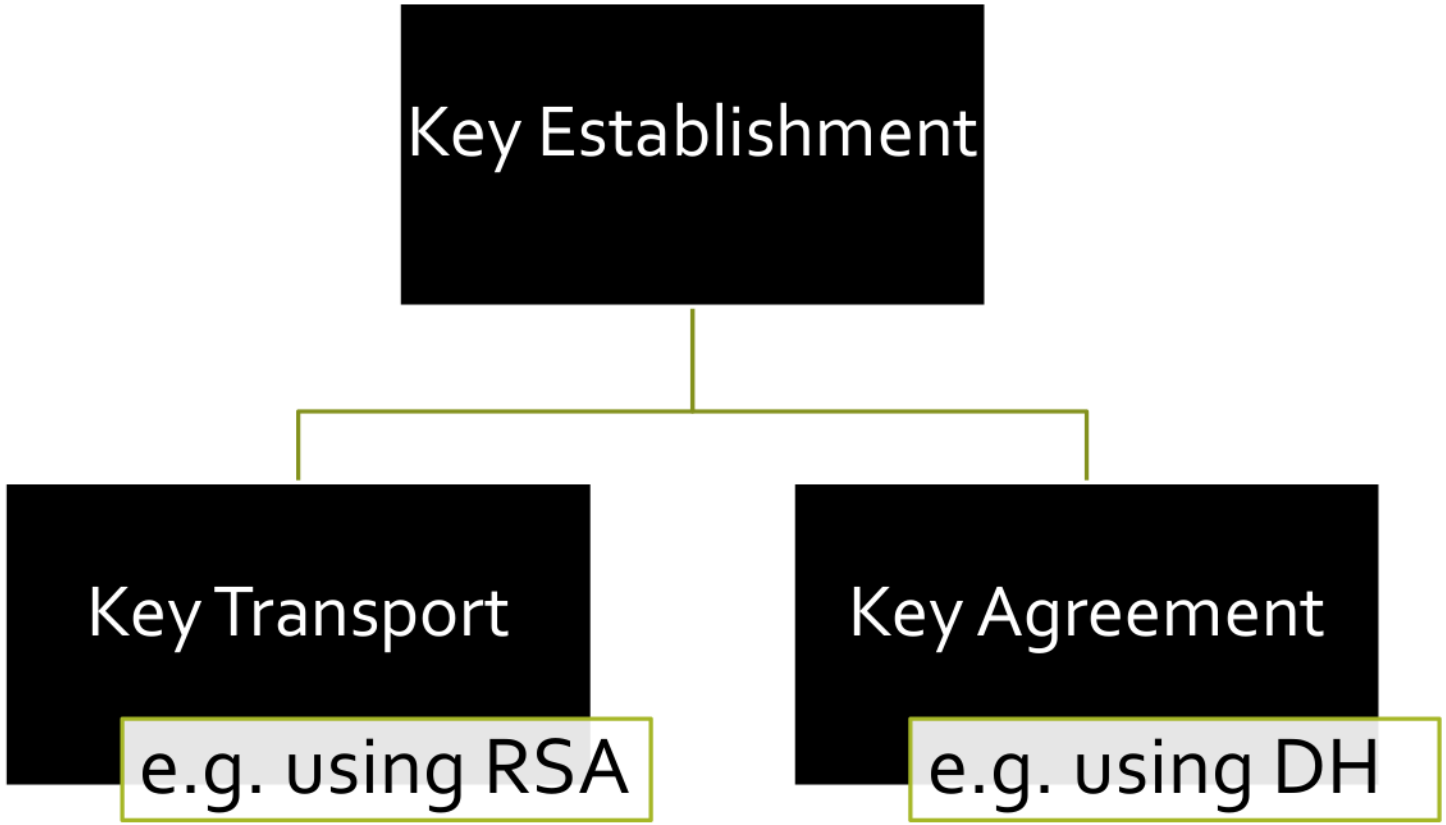
Practice

IP-18FKVKRBG

Lecture 10

2022-2023-2

Key Establishment Approaches



Diffie Hellmann

Diffie–Hellman I

Alice **Bob**

Common Parameters

Prime number p

Generator g

Choose random private key

$k_{prA} = a \in \{1,2,\ldots,p\text{-}1\}$

Choose random private key

$k_{prB} = b \in \{1,2,\ldots,p\text{-}1\}$

Diffie–Hellman II

Alice

…

Compute corresponding public key

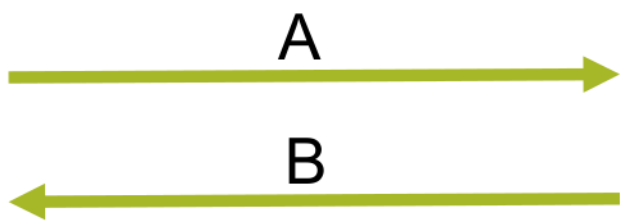
$A = g^a \bmod p$

Bob

…

Compute correspondig public key

$B = g^b \bmod p$



Compute common secret

$k_{AB} = B^a = (g^b)^a \bmod p$

Compute common secret

$k_{AB} = A^b = (g^a)^b \bmod p$

Diffie–Hellman III

Diffie- Hellman is only a key exchange protocol. The joined key can be used in symmetric encryption.

Alice **Bob**

$y = Enc(m, k_{AB})$ $m = Dec(y, k_{AB})$

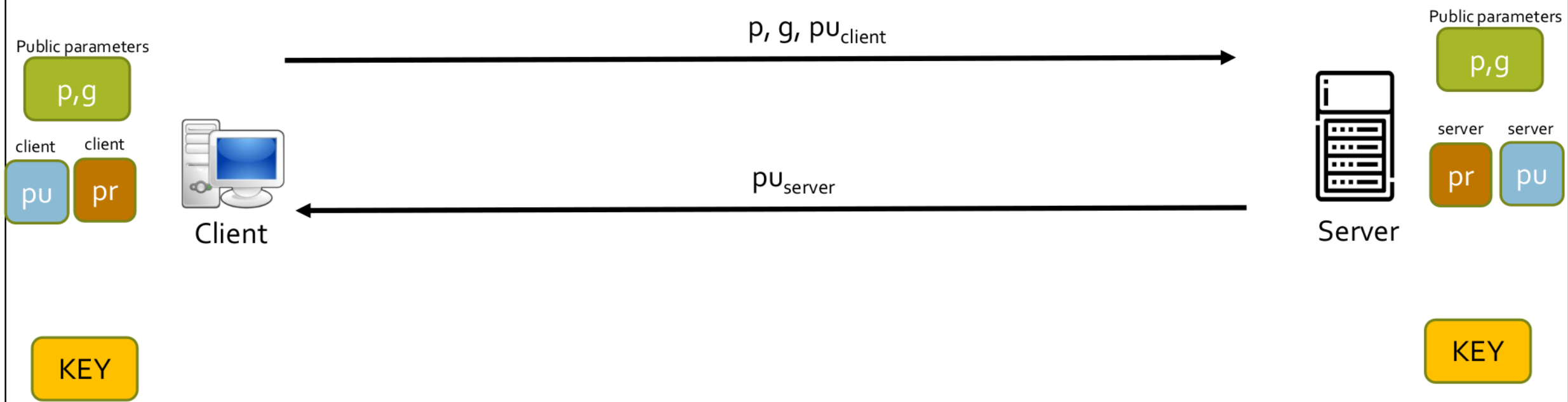
DH: Both parties in a local code

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
```

```
parameters = dh.generate_parameters(generator=2, key_size=512)

alice_private_key = parameters.generate_private_key()
bob_private_key = parameters.generate_private_key()
bob_public_key = bob_private_key.public_key()

alice_shared_key = alice_private_key.exchange(bob_public_key)

derived_key = HKDF(
    algorithm=hashes.SHA256(),
    length=32,
    salt=None,
    info=b'handshake data',
).derive(alice_shared_key)
```

1. Repeat the (shared key generation) and (key derivation) for the Bob as well.
2. Check whether both derived keys are identical?

DH: Client and Server



DH: Part 1 - Client sender side

Generates the public parameters, and its pair of keys

```
parameters = dh.generate_parameters(generator=2, key_size=512)
p = parameters.parameter_numbers().p
g = parameters.parameter_numbers().g
client_private_key = parameters.generate_private_key()
client_y = client_private_key.public_key().public_numbers().y
client_x = client_private_key.private_numbers().x
```

It stores (p, g, client_x and client_y) as the main parameters for later use.

Then, it sends (p, g and client_y) parameters to the server

DH: Part 2 - Server side

It first regenerate the same DH group, and the public key of the client, using the received parameters: p, g and client_y.

```
pn = dh.DHParameterNumbers(p,g)
client_public_number = dh.DHPublicNumbers(client_y, pn)
client_public_key = client_public_number.public_key()
```

Generates its key pair. Stores server_x to be fixed in later executions.

```
parameters = pn.parameters()
server_private_key = parameters.generate_private_key()
server_y = server_private_key.public_key().public_numbers().y
server_x = server_private_key.private_numbers().x
```

Then, it performs the shared key generation and key derivation.

DH: Part 3 - Client receiver side

It already stored the (p, g and client_y, client_x) parameters, and receives (server_y) from the server.

Regenerates its private key:

```
pn = dh.DHParameterNumbers(p,g)

client_public_number = dh.DHPublicNumbers(client_y, pn)
client_private_number = dh.DHPrivateNumbers(client_x, client_public_number)
client_private_key = client_private_number.private_key()
```

Regenerates the PK of the server:

```
server_public_number = dh.DHPublicNumbers(server_y, pn)
server_public_key = server_public_number.public_key()
```

Then it generate the shared key, and performs the key derivation.