Lec2_Cryptography I

What is crytography

confidentiality - symetric key encryption, public-key encryption

integrity: hash function, MAC, digital signature

data origin authentication: message originated source

entity authentication: know person is claimed one

non-repudiation

Symmetric encryption

Dec k(Enc k M) = M

hide: key + plaintext

Classic cipher

shift/ Ceasar cipher algorithm

(mod) - modulo

plaintext = D(key, ciphertext) = (ciphertext - key) mod 26

ciphertext = E(key, plaintext) = (plaintext + key) mod 26

ROT N(ROTATE)

https://www.dcode.fr/caesar-cipher

https://planetcalc.com/1434/

*security should depend only on the key

Brute force key search - frequency analysis

```
# Fix this encryption function, it does not encrypt correctly!
def encrypt(letter):
    return chr((ord(letter) - ord('A') +3) % 26 + ord('A'))
# Add a decription function!
def decrypt(letter):
    return chr((ord(letter)- ord('A') -3) % 26 + ord('A')) #fix

print(''.join([encrypt(l) for l in list("HELLO")]))
print(''.join([decrypt(l) for l in list("KHOOR")]))
```

Vernam cipher

XOR operator

Encryption: c(i) = m(i) ⊕ k(i)

Decryption: M⊕K⊕K = M

Vernam cipher use any bit sourse as key

OTP (One-Time-Pad)

Vernam cipher with random key (not reused)

Encryption: C = P ⊕ K

length of plaintext and key are same

1. decide key
2. send message (calculation using +)
3. decrypt (calculation using -)

https://www.boxentriq.com/code-breaking/one-time-pad

K is not reused

```
# One time pad cipher
# encryption and decryption
def encrypt(plaintext, key):
    ciphertext = ""
```

```
    key_length = len(key)
    for i in range(len(plaintext)):
        char = plaintext[i]
        key_char = key[i % key_length]
        encrypted_char = chr((ord(char) + ord(key_char)) % 26 +
        ciphertext += encrypted_char
    return ciphertext

def decrypt(ciphertext, key):
    plaintext = ""
    key_length = len(key)
    for i in range(len(ciphertext)):
        char = ciphertext[i]
        key_char = key[i % key_length]
        decrypted_char = chr((ord(char) - ord(key_char)) % 26 +
        plaintext += decrypted_char
    return plaintext

# Example usage
plaintext = "HELLOWORLD"
key = "POLYUROCKS"
ciphertext = encrypt(plaintext, key)
decrypted_text = decrypt(ciphertext, key)

print("Plaintext:", plaintext)
print("Ciphertext:", ciphertext)
print("Decrypted text:", decrypted_text)
```

Stream cipher

Block cipher: broken into fixed-length blocks before encryption, more modern, one block processed at a time

Stream cipher: block length is one bit/ char, require only limited buffering of data, letter by letter

Calculate Stream Cipher

Key Stream = Stream Cipher Algorithm (Key, Nonce)

Nonce - number used only once

C = P ⊕ KS

P = C ⊕ KS

*Counter based

Synchronization

Synchronous: keystream needs to be synchronized with the ciphertext

Self-synchronizing: key generated with function of key, fixed number of ciphertext digits

RC4 - code

```
# Class exercise
# Encrypt the string “We love crypto at PolyU!” using the key “s
# The output will not be printable in plain letters, encode it i
# The output should start with 0xB3

def initialize_state():
    state = bytearray(range(256))
    return state

def convert_key(key):
    return bytearray(key, 'utf-8')

def key_scheduling(state, key):
    key_length = len(key)
    j = 0

    for i in range(256):
        j = (j + state[i] + key[i % key_length]) % 256
        state[i], state[j] = state[j], state[i]
```

```
    return state

def generate_keystream(state):
    i = 0
    j = 0

    # iterate from 0 to 255
    while True:
        i = (i + 1) % 256
        j = (j + state[i]) % 256
        state[i], state[j] = state[j], state[i]
        yield state[(state[i] + state[j]) % 256]


def encrypt(plaintext, key):
    rc4_state = initialize_state()
    key = convert_key(key)
    rc4_state = key_scheduling(rc4_state, key)
    keystream = generate_keystream(rc4_state)

    ciphertext = bytearray()
    for char in plaintext.encode('utf-8'):
        keystream_byte = next(keystream)
        encrypted_byte = char ^ keystream_byte
        ciphertext.append(encrypted_byte)

    return "0x" + ciphertext.hex()

plaintext = "We love crypto at PolyU!"
key = "superlongkey"

encrypted_data = encrypt(plaintext, key)
print("Encrypted Data (Hex):", encrypted_data)
```

Randomness for keys

true randomness from analog event is difficult to collect and too slow

Solution

- combine with deterministic algorithm with true randomness
- PRNG(seed) = random-looking string
- seed is unpredictable, out is unpredictable too
- seed = electrical noise in computer

Cryptographically Secure

1. next-bit test: unpredictable of the next bit with past bits
2. balanced: number of 1, 0 should be equal
3. non-linearity

Computational security

level computation required is far outweigh the computational resources of adversary

use case

stream cipher - RC4

key prefer random / derived from password

Entropy

source:

1. CPU support - Intel RdRand / backdoored;
2. Online Services(random.org, not trusted)

3. Cloudflare

Not good for cryptographic

```
# C/C++
srand(time(NULL));
rand();

# Java
Random randomGenerator = Random();
int randomInt = randomGenerator.nextInt(100);

# Python
random.randint(1,10)
```

Good entropy sources

```
# C++
# use API
CryptGenRandom();

# Java
SecureRandom random = new SecureRandom();
random.nextBytes(bytes);

# Python
os.urandom(n)
```