Soumodipta Bose| 2021201086

Use the PRF in some secure mode of operation to obtain CPA Secure encryption scheme(Code)

CPA Encryption/Decryption:

- **encrypt(nonce, message, CHUNK_LENGTH, PRIVATE_KEY):**

Encrypts binary message in CPA secure encryption in OFB mode

Args:

nonce (binary string - n bits): (also called IV) random initial value taken only once

message (binary string): Message to be transmitted

CHUNK_LENGTH (int): Size of message fragment

PRIVATE_KEY (binary string - n bits): Private shared key

Returns:

r – initialization vector, will be used at decryption

c - binary string : Encrypted message

- **decrypt(r, cipher_text, CHUNK_LENGTH, PRIVATE_KEY):**

Decrypts the cipher text into plain text in OFB mode

Args:

r (binary string - n bits): Used for decryption

cipher_text (binary string): Cipher text to be deciphered

CHUNK_LENGTH (int): Size of message fragment

PRIVATE_KEY (binary string - n bits): Private shared key

Returns:

binary string : Decrypted message

Usage:

1. Create a binary string for private key ex. key='10001001' as key of 16 bits or more, note this key is used in PRF and needs to be available at both sender and receiver.
2. Take the message to be encrypted as a string.
3. Use **str_to_bin(s)** utility to convert message to binary.
4. Then select size of each chunk for fragmenting the message.
5. A one time nonce is generated using the utility function **get_random_bits(size)** to match the size of the message chunk, this will be used to generate the initialization vector.
6. Encrypt the binary message using **encrypt()**and send the parameters as per the specifications.
7. Both r and c has to be send to receiver.

8. Use **decrypt()** to decrypt the message and get the original message.
9. Then just use **bin_to_str(n)** to get the original string message back from binary.
10. The demo code in **start()** contains all the above steps.

Crypto library:

- **gen(x, p=doubler()):**Pseudo Random number generator

Args:

x (binary string): Initial Seed

p (function, optional): A polynomial input can be given. Defaults to doubler. The function can be anonymous function as well as long as it returns an integer and takes length of initial key as input

Returns: (binary string): Pseudo random bits

PRG_single(x) and **PRG_double(x)** are wrappers for **gen(x,p)** where former retains same length and latter doubles the length

- **F(k, x):**Pseudo random function

Args:

k (string): seed (n bits) binary string

x (string): input string (binary)

Returns: n bit truly random binary string

- **get_random_bits(size):** Returns random bits of length given by the parameter size, built using PRG and initial seed is taken from the Operating system time in millisecond.

Utility functions:

- **split_string(x):** splits string into two equal parts
- **dec_to_bin_wo_pad(x):** Converts decimal to binary without padding
- **dec_to_bin(x, size):**Converts decimal to binary with padding
- **bin_to_dec(x):** converts binary to decimal
- **discrete_log(x):**

DLP One way function GENERATOR = 8173 MOD = 65521

Performs (GENERATOR^x) % (MOD)

Args: x (int): seed value

Returns: one way function value

- **get_hardcore_bit(x):**

Extracts Hardcore bit using Blum Micali Hardcore bit

- **g(x):**Takes input binary string x(L bits) and return hardcore bit and new seed of L+1 bits. Calls discrete_log(x) and get_hardcore_bit(x)
- **xor(bin_x, bin_y):** returns xor of the two binary strings
- **str_to_bin(s):**Converts string into binary string
- **bin_to_str(n):**Converts binary string into normal string