**Министерство науки и высшего образования РФ**

Федеральное государственное бюджетное общеобразовательное учреждение высшего образования

«Чувашский Государственный Университет им. И.Н.Ульянова»

 

Факультет информатики и вычислительной техники

Кафедра компьютерных технологий

   
   
   
 

Отчет Лабораторной работы №1

По дисциплине: «Информационная безопасность»

Вариант 4

Выполнил студент группы КТ-43-21

Казаков А.Ю.

Проверил ст. преподаватель:

Мытникова Е. А.

   
   
   
   
   
   
 

Чебоксары, 2024

from functions.cbc import \*  
from functions.cfb import \*  
from functions.ecb import \*  
from functions.gcm import \*  
from functions.ofb import \*  
  
  
plaintext = "Hello, my super world!"  
key = "secretkey"  
iv = "randomiv"  
bin\_data = b'Hello, world!'  
  
cbc = cbc\_encrypt(plaintext, key, iv)  
cfb = cfb\_encrypt(plaintext, key, iv)  
ecb = ecb\_encrypt(plaintext, key)  
gcm = gcm\_encrypt(plaintext, key, iv)  
ofb = ofb\_encrypt(plaintext, key, iv)  
  
cbc\_bytes = cbc\_encrypt(bin\_data, key, iv)  
cfb\_bytes = cfb\_encrypt(bin\_data, key, iv)  
ecb\_bytes = ecb\_encrypt(bin\_data, key)  
gcm\_bytes = gcm\_encrypt(bin\_data, key, iv)  
ofb\_bytes = ofb\_encrypt(bin\_data, key, iv)  
  
  
def print\_table(title, data, is\_binary=False):  
 print(f"\n{title}\n")  
 # Заголовки таблицы  
 header = "| {:<8} | {:<80} |".format("Режим", "Зашифрованные данные")  
 separator = "-" \* len(header)  
 print(separator)  
 print(header)  
 print(separator)  
  
 # Данные для таблицы  
 for mode, value in data.items():  
 # Если это GCM, выводим кортеж (ciphertext, tag)  
 if isinstance(value, tuple):  
 value\_str = f"({repr(value[0])}, {repr(value[1])})"  
 else:  
 value\_str = repr(value)  
 print("| {:<8} | {:<80} |".format(mode, value\_str))  
 print(separator)  
  
  
text\_data = {  
 "CBC": cbc,  
 "CFB": cfb,  
 "ECB": ecb,  
 "GCM": gcm,  
 "OFB": ofb  
}  
  
binary\_data = {  
 "CBC": cbc\_bytes,  
 "CFB": cfb\_bytes,  
 "ECB": ecb\_bytes,  
 "GCM": gcm\_bytes,  
 "OFB": ofb\_bytes  
}  
  
print\_table(f"#1 Шифрование текста - \"{plaintext}\"; Ключ - {key}", text\_data)  
print\_table(f"#2 Шифрование бинарных данных - {bin\_data}; Ключ - {key}", binary\_data)

def pad\_text(text, block\_size):  
 *"""Дополняет текст до длины, кратной block\_size"""* padding\_len = block\_size - (len(text) % block\_size)  
 padding = bytes([padding\_len] \* padding\_len)  
 return text + padding  
  
def xor\_bytes(a, b):  
 *"""Выполняет побитовое XOR двух байтовых строк одинаковой длины"""* return bytes(x ^ y for x, y in zip(a, b))  
  
  
def cbc\_encrypt(plaintext, key, iv):  
 *"""Шифрование в режиме CBC (принимает как текст, так и бинарные данные)"""* block\_size = 8  
  
 # Если plaintext - строка, преобразуем в байты, иначе считаем, что это уже байты  
 if isinstance(plaintext, str):  
 plaintext\_bytes = plaintext.encode('utf-8')  
 elif isinstance(plaintext, bytes):  
 plaintext\_bytes = plaintext  
 else:  
 raise ValueError("plaintext должен быть строкой или байтами")  
  
 key\_bytes = key.encode('utf-8')  
 iv\_bytes = iv.encode('utf-8')  
  
 if len(key\_bytes) < block\_size:  
 key\_bytes = key\_bytes + b'\x00' \* (block\_size - len(key\_bytes))  
 key\_bytes = key\_bytes[:block\_size]  
  
 if len(iv\_bytes) < block\_size:  
 iv\_bytes = iv\_bytes + b'\x00' \* (block\_size - len(iv\_bytes))  
 iv\_bytes = iv\_bytes[:block\_size]  
  
 # Дополнение данных  
 padded\_data = pad\_text(plaintext\_bytes, block\_size)  
  
 ciphertext = b''  
 previous\_block = iv\_bytes  
  
 for i in range(0, len(padded\_data), block\_size):  
 block = padded\_data[i:i + block\_size]  
 block\_xor = xor\_bytes(block, previous\_block)  
 encrypted\_block = xor\_bytes(block\_xor, key\_bytes)  
 ciphertext += encrypted\_block  
 previous\_block = encrypted\_block  
  
 return ciphertext

def xor\_bytes(a, b):  
 *"""Выполняет побитовое XOR двух байтовых строк"""* return bytes(x ^ y for x, y in zip(a, b))  
  
  
def cfb\_encrypt(plaintext, key, iv, segment\_size=1):  
 *"""Шифрование в режиме CFB"""* block\_size = 8  
  
 if isinstance(plaintext, str):  
 plaintext\_bytes = plaintext.encode('utf-8')  
 elif isinstance(plaintext, bytes):  
 plaintext\_bytes = plaintext  
 else:  
 raise ValueError("plaintext должен быть строкой или байтами")  
  
 key\_bytes = key.encode('utf-8')  
 iv\_bytes = iv.encode('utf-8')  
  
 if len(key\_bytes) < block\_size:  
 key\_bytes = key\_bytes + b'\x00' \* (block\_size - len(key\_bytes))  
 key\_bytes = key\_bytes[:block\_size]  
   
 if len(iv\_bytes) < block\_size:  
 iv\_bytes = iv\_bytes + b'\x00' \* (block\_size - len(iv\_bytes))  
 iv\_bytes = iv\_bytes[:block\_size]  
   
 ciphertext = b''  
 shift\_register = iv\_bytes  
  
 for i in range(0, len(plaintext\_bytes), segment\_size):  
 encrypted\_block = xor\_bytes(shift\_register, key\_bytes)  
 plaintext\_segment = plaintext\_bytes[i:i + segment\_size]  
 ciphertext\_segment = xor\_bytes(plaintext\_segment, encrypted\_block[:segment\_size])  
 ciphertext += ciphertext\_segment  
  
 shift\_register = shift\_register[segment\_size:] + ciphertext\_segment  
   
 return ciphertext

def pad\_text(text, block\_size):  
 *"""Дополняет текст до длины, кратной block\_size"""* padding\_len = block\_size - (len(text) % block\_size)  
 padding = bytes([padding\_len] \* padding\_len)  
 return text + padding  
  
  
def ecb\_encrypt(plaintext, key):  
 *"""Шифрование в режиме ECB"""* block\_size = 8  
 # Если plaintext - строка, преобразуем в байты, иначе считаем, что это уже байты  
 if isinstance(plaintext, str):  
 plaintext\_bytes = plaintext.encode('utf-8')  
 elif isinstance(plaintext, bytes):  
 plaintext\_bytes = plaintext  
 else:  
 raise ValueError("plaintext должен быть строкой или байтами")  
  
 key\_bytes = key.encode('utf-8')  
  
 if len(key\_bytes) < block\_size:  
 key\_bytes = key\_bytes + b'\x00' \* (block\_size - len(key\_bytes))  
 key\_bytes = key\_bytes[:block\_size]  
  
 padded\_text = pad\_text(plaintext\_bytes, block\_size)  
  
 ciphertext = b''  
 for i in range(0, len(padded\_text), block\_size):  
 block = padded\_text[i:i + block\_size]  
 encrypted\_block = bytes(a ^ b for a, b in zip(block, key\_bytes))  
 ciphertext += encrypted\_block  
  
 return ciphertext

def xor\_bytes(a, b):  
 *"""Выполняет побитовое XOR двух байтовых строк"""* return bytes(x ^ y for x, y in zip(a, b))  
  
def inc\_counter(counter):  
 *"""Увеличивает 32-битный счётчик в конце блока"""* counter\_int = int.from\_bytes(counter[-4:], 'big')  
 counter\_int = (counter\_int + 1) & 0xFFFFFFFF # Ограничиваем 32 битами  
 return counter[:-4] + counter\_int.to\_bytes(4, 'big')  
  
def ghash(h, data):  
 *"""Упрощённая имитация GHASH (XOR вместо умножения в поле Галуа)"""* block\_size = 16  
 result = bytes(block\_size)  
  
 if len(data) % block\_size != 0:  
 data += b'\x00' \* (block\_size - len(data) % block\_size)  
  
 for i in range(0, len(data), block\_size):  
 block = data[i:i + block\_size]  
 result = xor\_bytes(result, block)  
   
 return xor\_bytes(result, h)  
  
def gcm\_encrypt(plaintext, key, iv, aad=b''):  
 *"""Шифрование в режиме GCM"""* block\_size = 16  
  
 if isinstance(plaintext, str):  
 plaintext\_bytes = plaintext.encode('utf-8')  
 elif isinstance(plaintext, bytes):  
 plaintext\_bytes = plaintext  
 else:  
 raise ValueError("plaintext должен быть строкой или байтами")  
  
 key\_bytes = key.encode('utf-8')  
 iv\_bytes = iv.encode('utf-8')  
  
 if len(key\_bytes) < block\_size:  
 key\_bytes = key\_bytes + b'\x00' \* (block\_size - len(key\_bytes))  
 key\_bytes = key\_bytes[:block\_size]  
  
 if len(iv\_bytes) != 12:  
 iv\_bytes = iv\_bytes.ljust(12, b'\x00')[:12]  
 counter = iv\_bytes + b'\x00\x00\x00\x01'  
  
 h = xor\_bytes(bytes(block\_size), key\_bytes)  
   
 ciphertext = b''  
 for i in range(0, len(plaintext\_bytes), block\_size):  
 keystream = xor\_bytes(counter, key\_bytes)  
 plaintext\_block = plaintext\_bytes[i:i + block\_size]  
 ciphertext\_block = xor\_bytes(plaintext\_block, keystream[:len(plaintext\_block)])  
 ciphertext += ciphertext\_block  
 counter = inc\_counter(counter)  
  
 len\_aad = len(aad) \* 8  
 len\_ciphertext = len(ciphertext) \* 8  
 auth\_data = aad + ciphertext + len\_aad.to\_bytes(8, 'big') + len\_ciphertext.to\_bytes(8, 'big')  
 tag = ghash(h, auth\_data)  
   
 return ciphertext, tag

def xor\_bytes(a, b):  
 *"""Выполняет побитовое XOR двух байтовых строк"""* return bytes(x ^ y for x, y in zip(a, b))  
  
def ofb\_encrypt(plaintext, key, iv):  
 *"""Шифрование в режиме OFB"""* block\_size = 8  
  
 if isinstance(plaintext, str):  
 plaintext\_bytes = plaintext.encode('utf-8')  
 elif isinstance(plaintext, bytes):  
 plaintext\_bytes = plaintext  
 else:  
 raise ValueError("plaintext должен быть строкой или байтами")  
  
 key\_bytes = key.encode('utf-8')  
 iv\_bytes = iv.encode('utf-8')  
  
 if len(key\_bytes) < block\_size:  
 key\_bytes = key\_bytes + b'\x00' \* (block\_size - len(key\_bytes))  
 key\_bytes = key\_bytes[:block\_size]  
   
 if len(iv\_bytes) < block\_size:  
 iv\_bytes = iv\_bytes + b'\x00' \* (block\_size - len(iv\_bytes))  
 iv\_bytes = iv\_bytes[:block\_size]  
   
 ciphertext = b''  
 shift\_register = iv\_bytes  
  
 for i in range(len(plaintext\_bytes)):  
 keystream\_block = xor\_bytes(shift\_register, key\_bytes)  
 plaintext\_byte = plaintext\_bytes[i:i + 1]  
 ciphertext\_byte = xor\_bytes(plaintext\_byte, keystream\_block[:1])  
 ciphertext += ciphertext\_byte  
  
 shift\_register = keystream\_block  
   
 return ciphertext

