Белорусский государственный университет информатики и радиоэлектроники

Кафедра информатики

Лабораторная работа № 2

Симметричная криптография. СТБ 34.101.31-2011.

Выполнила студент гр. 653502: Серебренников В.А.

Проверил ассистент КИ: Артемьев В. С.

Минск, 2019

Введение

Настоящий стандарт определяет семейство криптографических алгоритмов, предназначенных для обеспечения конфиденциальности и контроля целостности данных. Обрабатываемыми данными являются двоичные слова (сообщения).

Криптографические алгоритмы стандарта построены на основе базовых алгоритмов шифрования блока данных. Криптографические алгоритмы шифрования и контроля целостности делятся на восемь групп:

1) алгоритмы шифрования в режиме простой замены;
2) алгоритмы шифрования в режиме сцепления блоков;
3) алгоритмы шифрования в режиме гаммирования с обратной связью;
4) алгоритмы шифрования в режиме счетчика;
5) алгоритм выработки имитовставки;
6) алгоритмы одновременного шифрования и имитозащиты данных;
7) алгоритмы одновременного шифрования и имитозащиты ключа;
8) алгоритм хэширования.

Первые четыре группы предназначены для обеспечения конфиденциальности сообщений. Каждая группа включает алгоритм зашифрования и алгоритм расшифрования. Стороны, располагающие общим ключом, могут организовать конфиденциальный обмен сообщениями путем их шифрования перед отправкой и расшифрования после получения. В режимах простой замены и сцепления блоков шифруются сообщения, которые содержат хотя бы один блок, а в режимах гаммирования с обратной связью и счетчика — сообщения произвольной длины.

В рамках лабораторной работы необходимо реализовать программные средства шифрования и дешифрования при помощи алгоритма СТБ 34.101.31-2011 в различных режимах.

Блок-схема алгоритма

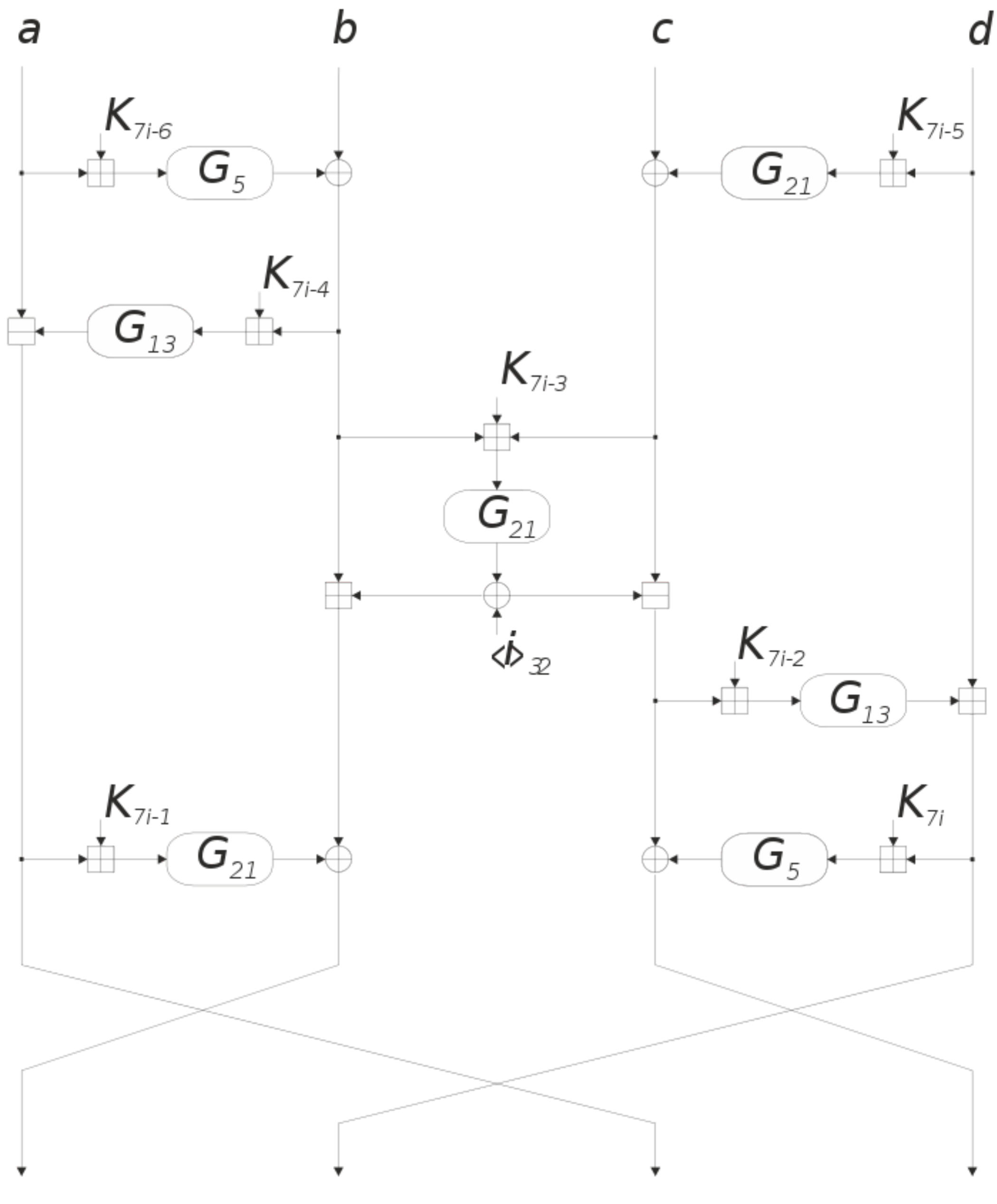

Рис.1. Вычисления на i-ом такте шифрования.

Пример выполнения программы

```
encrypted result:   §4ô¿£¢0qdÖAYô□ëö-□ô'T□ªgÖ!É^èóÉ
decrypted result:  qwertyuioplkjhgfdsazxcvbnmkjhytr
encrypted result:  Ü¯Z'çcgÌ|Åéó□Çó¾Á□íÇÓÕê‰□□ÁÄ□ÂÔÄ
decrypted result:  qwertyuioplkjhgfdsazxcvbnmkjhytr
```

Рис. 2. Пример работы программы

Код программы

```
class Converter:

    @classmethod
    def to_bin(cls, data):
        array = list()
        for char in data:
            binval = cls.binvalue(char, 8)  # Get the char value on one byte
            array.extend([int(x) for x in list(binval)])
        return array

    @classmethod
    def binvalue(cls, val, bitsize):  # Return the binary value as a string of the
given size
        binval = bin(val)[2:] if isinstance(val, int) else bin(ord(val))[2:]
        while len(binval) < bitsize:
            binval = "0" + binval
        return binval

    @classmethod
    def to_string(cls, data):
        chars = []
        for i in range(len(data) // 8):
            byte = data[i * 8:(i + 1) * 8]
            byte_str = ''.join(map(str, byte))
            chars.append(chr(int(byte_str, 2)))
        return ''.join(chars)

    @classmethod
    def to_int(cls, value):
        return int(''.join(map(str, value)), 2)

    @classmethod
    def int_to_bits(cls, n, bits_count):
        bits = []
        for digit in bin(n)[2:]:
            bits.append(int(digit))
        while len(bits) < bits_count:
            bits.insert(0, 0)
        return bits

from converter import Converter

class Operation:

    @classmethod
    def plus_mod_32(cls, a, b):
        int_a = Converter.to_int(a)
        int_b = Converter.to_int(b)
        int_result = (int_a + int_b) % (2 ** 32)
        result = Converter.int_to_bits(int_result, 32)
        return result

    @classmethod
    def minus_mod_32(cls, a, b):
```

```
        int_a = Converter.to_int(a)
        int_b = Converter.to_int(b)
        int_result = (int_a - int_b) % (2 ** 32)
        result = Converter.int_to_bits(int_result, 32)
        return result

    @classmethod
    def bit_xor(cls, arr1, arr2):
        bit_s = []
        for index, item in enumerate(arr1):
            bit_s.append(arr1[index] ^ arr2[index])
        return bit_s

import copy
from operation import Operation
from converter import Converter

class STB:

    __H_TABLE = [
        [0xB1, 0x94, 0xBA, 0xC8, 0x0A, 0x08, 0xF5, 0x3B, 0x36, 0x6D, 0x00, 0x8E,
0x58, 0x4A, 0x5D, 0xE4],
        [0x85, 0x04, 0xFA, 0x9D, 0x1B, 0xB6, 0xC7, 0xAC, 0x25, 0x2E, 0x72, 0xC2,
0x02, 0xFD, 0xCE, 0x0D],
        [0x5B, 0xE3, 0xD6, 0x12, 0x17, 0xB9, 0x61, 0x81, 0xFE, 0x67, 0x86, 0xAD,
0x71, 0x6B, 0x89, 0x0B],
        [0x5C, 0xB0, 0xC0, 0xFF, 0x33, 0xC3, 0x56, 0xB8, 0x35, 0xC4, 0x05, 0xAE,
0xD8, 0xE0, 0x7F, 0x99],
        [0xE1, 0x2B, 0xDC, 0x1A, 0xE2, 0x82, 0x57, 0xEC, 0x70, 0x3F, 0xCC, 0xF0,
0x95, 0xEE, 0x8D, 0xF1],
        [0xC1, 0xAB, 0x76, 0x38, 0x9F, 0xE6, 0x78, 0xCA, 0xF7, 0xC6, 0xF8, 0x60,
0xD5, 0xBB, 0x9C, 0x4F],
        [0xF3, 0x3C, 0x65, 0x7B, 0x63, 0x7C, 0x30, 0x6A, 0xDD, 0x4E, 0xA7, 0x79,
0x9E, 0xB2, 0x3D, 0x31],
        [0x3E, 0x98, 0xB5, 0x6E, 0x27, 0xD3, 0xBC, 0xCF, 0x59, 0x1E, 0x18, 0x1F,
0x4C, 0x5A, 0xB7, 0x93],
        [0xE9, 0xDE, 0xE7, 0x2C, 0x8F, 0x0C, 0x0F, 0xA6, 0x2D, 0xDB, 0x49, 0xF4,
0x6F, 0x73, 0x96, 0x47],
        [0x06, 0x07, 0x53, 0x16, 0xED, 0x24, 0x7A, 0x37, 0x39, 0xCB, 0xA3, 0x83,
0x03, 0xA9, 0x8B, 0xF6],
        [0x92, 0xBD, 0x9B, 0x1C, 0xE5, 0xD1, 0x41, 0x01, 0x54, 0x45, 0xFB, 0xC9,
0x5E, 0x4D, 0x0E, 0xF2],
        [0x68, 0x20, 0x80, 0xAA, 0x22, 0x7D, 0x64, 0x2F, 0x26, 0x87, 0xF9, 0x34,
0x90, 0x40, 0x55, 0x11],
        [0xBE, 0x32, 0x97, 0x13, 0x43, 0xFC, 0x9A, 0x48, 0xA0, 0x2A, 0x88, 0x5F,
0x19, 0x4B, 0x09, 0xA1],
        [0x7E, 0xCD, 0xA4, 0xD0, 0x15, 0x44, 0xAF, 0x8C, 0xA5, 0x84, 0x50, 0xBF,
0x66, 0xD2, 0xE8, 0x8A],
        [0xA2, 0xD7, 0x46, 0x52, 0x42, 0xA8, 0xDF, 0xB3, 0x69, 0x74, 0xC5, 0x51,
0xEB, 0x23, 0x29, 0x21],
        [0xD4, 0xEF, 0xD9, 0xB4, 0x3A, 0x62, 0x28, 0x75, 0x91, 0x14, 0x10, 0xEA,
0x77, 0x6C, 0xDA, 0x1D]]


    def __init__(self, key):
        key = Converter.to_bin(key)
```

```python
        sub_keys = self.__split_into_sub_lists(key, 32)  # 8 ключей по 32 бита
        self.sub_keys = sub_keys * 7  # 56 тактовых ключей по 32 бита

    def get_data_from_table(self, value):
        if '0x' in value:
            value = value[2:]
            if len(value) == 1:
                i = int('0x0', 16)
                j = int(value[0], 16)
                return self.__H_TABLE[i][j]
        i = int(value[0], 16)
        j = int(value[1], 16)
        return self.__H_TABLE[i][j]

    def check_text_length(self, data):
        if len(data) % 16 != 0:
            raise ValueError('Wrong length of text')

    def G_operation(self, data, r):
        splitted = self.__split_into_sub_lists(data, 8)
        result = []
        for value in splitted:
            int_val = Converter.to_int(value)
            hex_val = str(hex(int_val))
            result.append(self.get_data_from_table(hex_val))

        for i in range(len(result)):
            result[i] = Converter.int_to_bits(result[i], 8)

        result = [item for sublist in result for item in sublist]
        result = self.__left_shift(result, r)
        while len(result) < 32:
            result.insert(0, 0)
        return result

    def __left_shift(self, data, n):
        return data[n:] + data[:n]

    def __split_into_sub_lists(self, data, size):
        arr = []
        for i in range(0, len(data), size):
            arr.append(data[i:i + size])
        return arr

    def __encrypt(self, text):
        bits_text = Converter.to_bin(text)  # 16 байт (блок) -> 128 бит
        words = self.__split_into_sub_lists(bits_text, 32)
        a = words[0]
        b = words[1]
        c = words[2]
        d = words[3]

        for i in range(1, 9):
            # 1
            sub_key = copy.deepcopy(self.sub_keys[7 * i - 6 - 1])
            value = Operation.plus_mod_32(a, sub_key)
```

```
            value = self.G_operation(value, 5)
            b = Operation.bit_xor(b, value)
            # 2
            sub_key = copy.deepcopy(self.sub_keys[7 * i - 5 - 1])
            value = Operation.plus_mod_32(d, sub_key)
            value = self.G_operation(value, 21)
            c = Operation.bit_xor(c, value)
            # 3
            sub_key = copy.deepcopy(self.sub_keys[7 * i - 4 - 1])
            value = Operation.plus_mod_32(b, sub_key)
            value = self.G_operation(value, 13)
            a = Operation.minus_mod_32(a, value)
            # 4
            sub_key = copy.deepcopy(self.sub_keys[7 * i - 3 - 1])
            value = Operation.plus_mod_32(b, c)
            value = Operation.plus_mod_32(value, sub_key)
            value = self.G_operation(value, 21)
            value_i = i % (2 ** 32)
            value_i = Converter.int_to_bits(value_i, 32)
            while len(value_i) < 32:
                value_i.insert(0, 0)
            e = Operation.bit_xor(value, value_i)
            # 5
            b = Operation.plus_mod_32(b, e)
            # 6
            c = Operation.minus_mod_32(c, e)
            # 7
            sub_key = copy.deepcopy(self.sub_keys[7 * i - 2 - 1])
            value = Operation.plus_mod_32(c, sub_key)
            value = self.G_operation(value, 13)
            d = Operation.plus_mod_32(d, value)
            # 8
            sub_key = copy.deepcopy(self.sub_keys[7 * i - 1 - 1])
            value = Operation.plus_mod_32(a, sub_key)
            value = self.G_operation(value, 21)
            b = Operation.bit_xor(b, value)
            # 9
            sub_key = copy.deepcopy(self.sub_keys[7 * i - 1])
            value = Operation.plus_mod_32(d, sub_key)
            value = self.G_operation(value, 5)
            c = Operation.bit_xor(c, value)
            # 10
            a, b = b, a
            # 11
            c, d = d, c
            # 12
            b, c = c, b

        encrypted = b + d + a + c
        encrypted = Converter.to_string(encrypted)
        return encrypted

    def __decrypt(self, text):
        bits_text = Converter.to_bin(text)
        words = self.__split_into_sub_lists(bits_text, 32)
        a = words[0]
```

```
        b = words[1]
        c = words[2]
        d = words[3]

        for i in range(8, 0, -1):
            # 1
            sub_key = copy.deepcopy(self.sub_keys[7 * i - 1])
            temp = Operation.plus_mod_32(a, sub_key)
            temp = self.G_operation(temp, 5)
            b = Operation.bit_xor(b, temp)
            # 2
            sub_key = copy.deepcopy(self.sub_keys[7 * i - 1 - 1])
            temp = Operation.plus_mod_32(d, sub_key)
            temp = self.G_operation(temp, 21)
            c = Operation.bit_xor(c, temp)
            # 3
            sub_key = copy.deepcopy(self.sub_keys[7 * i - 2 - 1])
            temp = Operation.plus_mod_32(b, sub_key)
            temp = self.G_operation(temp, 13)
            a = Operation.minus_mod_32(a, temp)
            # 4
            sub_key = copy.deepcopy(self.sub_keys[7 * i - 3 - 1])
            temp = Operation.plus_mod_32(b, c)
            temp = Operation.plus_mod_32(temp, sub_key)
            temp = self.G_operation(temp, 21)
            value_i = i % (2 ** 32)
            value_i = Converter.int_to_bits(value_i, 32)
            e = Operation.bit_xor(temp, value_i)
            # 5
            b = Operation.plus_mod_32(b, e)
            # 6
            c = Operation.minus_mod_32(c, e)
            # 7
            sub_key = copy.deepcopy(self.sub_keys[7 * i - 4 - 1])
            temp = Operation.plus_mod_32(c, sub_key)
            temp = self.G_operation(temp, 13)
            d = Operation.plus_mod_32(d, temp)
            # 8
            sub_key = copy.deepcopy(self.sub_keys[7 * i - 5 - 1])
            temp = Operation.plus_mod_32(a, sub_key)
            temp = self.G_operation(temp, 21)
            b = Operation.bit_xor(b, temp)
            # 9
            sub_key = copy.deepcopy(self.sub_keys[7 * i - 6 - 1])
            temp = Operation.plus_mod_32(d, sub_key)
            temp = self.G_operation(temp, 5)
            c = Operation.bit_xor(c, temp)
            # 10
            a, b = b, a
            # 11
            c, d = d, c
            # 12
            a, d = d, a

        decoded = c + a + d + b
        decoded = Converter.to_string(decoded)
```

```
        return decoded

    def encrypt_simple_substitute(self, data):
        self.check_text_length(data)
        encoded = []
        blocks = self.__split_into_sub_lists(data, 16)
        for block in blocks:
            block = self.__encrypt(block)
            encoded.append(block)

        encoded = ''.join(encoded)
        return encoded

    def decrypt_simple_substitute(self, data):
        self.check_text_length(data)
        decoded = []
        blocks = self.__split_into_sub_lists(data, 16)
        for block in blocks:
            decoded.append(self.__decrypt(block))

        decoded = ''.join(decoded )
        return decoded

    def encrypt_clutch_blocks(self, data, sync):
        self.check_text_length(data)
        encoded = []
        blocks = self.__split_into_sub_lists(data, 16)
        encoded_sync = self.__encrypt(sync)
        for block in blocks:
            res =
Converter.to_string(Operation.bit_xor(Converter.to_bin(encoded_sync),
Converter.to_bin(block)))
            encoded_part = self.__encrypt(res)
            encoded_sync = encoded_part
            encoded.append(encoded_part)

        encoded = ''.join(encoded)
        return encoded

    def decrypt_clutch_blocks(self, data, sync):
        self.check_text_length(data)
        decoded = []
        blocks = self.__split_into_sub_lists(data, 16)
        temp = self.__encrypt(sync)
        for block in blocks:
            bit_temp = Converter.to_bin(temp)
            ans =
Converter.to_string(Operation.bit_xor(Converter.to_bin(self.__decrypt(block)),
bit_temp))
            decoded.append(ans)
            temp = block

        decoded = ''.join(decoded)
        return decoded

    def encrypt_gamming(self, data, sync):
```

```
        blocks = self.__split_into_sub_lists(data, 16)
        encoded = []
        temp = sync
        for block in blocks:
            enc_temp = self.__encrypt(temp)
            enc_sliced = enc_temp[:len(block)]
            ans = Converter.to_string(Operation.bit_xor(Converter.to_bin(block),
Converter.to_bin(enc_sliced)))
            encoded.append(ans)
            temp = ans

        encoded = ''.join(encoded)
        return encoded

    def decrypt_gamma_with_feedback(self, data, sync):
        blocks = self.__split_into_sub_lists(data, 16)
        decoded = []
        temp = sync

        for block in blocks:
            enc_temp = self.__encrypt(temp)
            enc_sliced = enc_temp[:len(block)]
            ans = Converter.to_string(Operation.bit_xor(Converter.to_bin(block),
Converter.to_bin(enc_sliced)))
            decoded.append(ans)
            temp = block

        decoded = ''.join(decoded)
        return decoded

from stb import STB

key = 'RTYBHncnfeicnjiujUFCTYU234huU-sQ'

if __name__ == '__main__':
    stb = STB(key)
    data = input('Input Text: ')
    print()


    print('Шифрование в режиме простой замены')
    encrypted = stb.encrypt_simple_substitute(data)
    print("Encrypted ", encrypted)
    decrypted = stb.decrypt_simple_substitute(encrypted)
    print("Decrypted ", decrypted)

    print('\nШифрование в режиме сцепления блоков')
    sync = '12345678abcdefgh'

    encrypted = stb.encrypt_clutch_blocks(data, sync)
    print("Encrypted ", encrypted)
    decrypted = stb.decrypt_clutch_blocks(encrypted, sync)
    print("Decrypted ", decrypted)

    print('\nШифрование в режиме гаммирования с обратной связью')
    sync = '12345678abcdefgh'
```

```python
encrypted = stb.encrypt_gamming(data, sync)
print("Encrypted ", encrypted)
decrypted = stb.decrypt_gamma_with_feedback(encrypted, sync)
print("Decrypted ", decrypted)
```

Вывод

Настоящий стандарт определяет семейство криптографических алгоритмов шифрования и контроля целостности, которые используются для защиты информации при ее хранении, передаче и обработке. Настоящий стандарт применяется при разработке средств криптографической защиты информации.