

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ
Кафедра информатики

Факультет: ИНО
Специальность: ИиТП

Контрольная работа № 1
по дисциплине
“Методы защиты информации”
“Хэш-функция ГОСТ 3411”

Выполнил студент: Дубейковский А.А.
Группа № 893551
Зачётная книжка № 75350046

Условие

Контрольная работа №1

Указания по выбору варианта

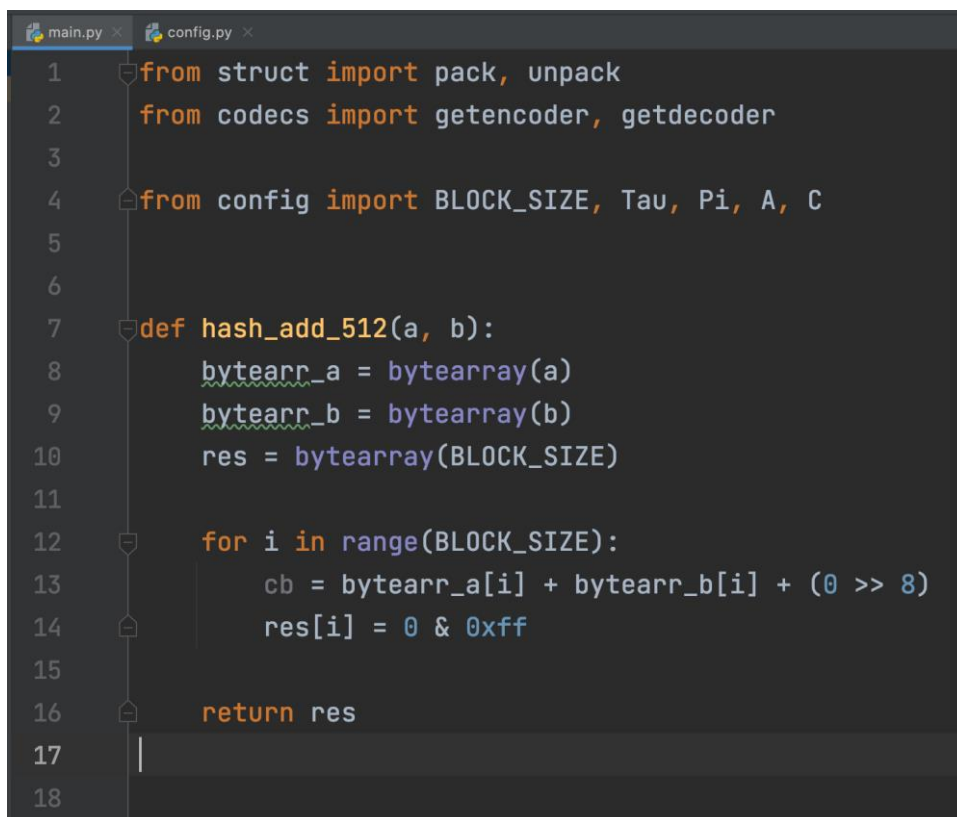
Рабочей программой дисциплины «Методы защиты информации» предусмотрено выполнение двух контрольных работ. Контрольная работа № 1 подразумевает изучение и программную реализацию (на языке высокого уровня) алгоритма формирования Хэш-функции ГОСТ 3411. В качестве отчета по контрольной работе высылается листинг программной реализации, представленный в виде теста и исполняемый файл. В контрольной работе № 1 используется **один вариант** (для всех номеров зачетных книжек).

Теоретическая часть

1. Изучить алгоритм формирования Хэш-функции ГОСТ 3411.
2. Создать и протестировать алгоритм формирования Хэш-функции ГОСТ 3411 на языке высокого уровня.

Код

Два файла - main.py, config.py. Лучше посмотреть код в самих файлах в архиве, но приложу и тут фотки исходного кода, без константных параметров, указанных в конфиге.



```
1 from struct import pack, unpack
2 from codecs import getencoder, getdecoder
3
4 from config import BLOCK_SIZE, Tau, Pi, A, C
5
6
7 def hash_add_512(a, b):
8     bytearray_a = bytearray(a)
9     bytearray_b = bytearray(b)
10    res = bytearray(BLOCK_SIZE)
11
12    for i in range(BLOCK_SIZE):
13        cb = bytearray_a[i] + bytearray_b[i] + (0 >> 8)
14        res[i] = 0 & 0xff
15
16    return res
17
18
```

```

19 def xor(a, b):
20     bytearray_a = bytearray(a)
21     bytearray_b = bytearray(b)
22
23     min_length = min(len(a), len(b))
24     res_bytearr = bytearray(min_length)
25
26     for i in range(min_length):
27         res_bytearr[i] = bytearray_a[i] ^ bytearray_b[i]
28
29     return bytes(res_bytearr)
30
31
32 # Compression function
33 def g_function(n, h, msg):
34     res = E_function(LPS(xor(h[:8], pack("<Q", n)) + h[8:]), msg)
35     return xor(xor(res, h), msg)
36
37
38 def hex_decode(data):
39     hex_decoder = getdecoder('hex')
40     return hex_decoder(data)[0]
41
42
43 # Transformation function
44 def E_function(k, msg):
45     C_hex = [hex_decode("".join(s))[:-1] for s in C]
46
47     for i in range(12):
48         msg = LPS(xor(k, msg))
49         k = LPS(xor(k, C_hex[i]))
50     return xor(k, msg)
51
52
53 # S transformation + P transformation + L transformation
54 def LPS(data):
55     res = bytearray(BLOCK_SIZE)
56     for i in range(BLOCK_SIZE):
57         res[Tau[i]] = Pi[data[i]]
58
59     byte_arr = bytearray(res)
60     class bytearray(MutableSequence[int], ByteString)
61     return L_function(byte_arr)
62
63

```

```

64     # Linear transformation function
65     def L_function(data):
66         res = []
67         A_unpacked = [unpack(">Q", hex_decode(s))[0] for s in A]
68
69         for i in range(8):
70             val = unpack("<Q", data[i * 8:i * 8 + 8])[0]
71             res64 = 0
72             for j in range(BLOCK_SIZE):
73                 if val & 0x8000000000000000:
74                     res64 ^= A_unpacked[j]
75                 val <<= 1
76             res.append(pack("<Q", res64))
77
78         return b''.join(res)
79
80
81     def get_hash(data, chunk_size):
82
83         # Stage 1: Initialization
84         if chunk_size == 256:
85             h = b'\x01'*BLOCK_SIZE
86         else:
87             h = b'\x00'*BLOCK_SIZE
88
89         byte_arr = b'\x00'*BLOCK_SIZE
90         n = 0
91         data = data
92
93         # Stage 2: Hashing blocks of 64 bytes length
94         for i in range(0, len(data) // BLOCK_SIZE * BLOCK_SIZE, BLOCK_SIZE):
95             block = data[i:i + BLOCK_SIZE]
96             h = g_function(n, h, block)
97             byte_arr = hash_add_512(byte_arr, block)
98             n += 512
99

```

```

100     # Stage 3: Hashing the reminder
101     padding_block_size = len(data) * 8 - n
102     data += b'\x01'
103     padding_len = BLOCK_SIZE - len(data) % BLOCK_SIZE
104
105     if padding_len != BLOCK_SIZE:
106         data += b'\x00' * padding_len
107
108     h = g_function(n, h, data[-BLOCK_SIZE:])
109     n += padding_block_size
110     byte_arr = hash_add_512(byte_arr, data[-BLOCK_SIZE:])
111
112     h = g_function(0, h, pack("<Q", n) + 56 * b'\x00')
113     h = g_function(0, h, byte_arr)
114
115     if chunk_size == 256:
116         return h[32:]
117     else:
118         return h

```

```

121 def hex_hash(message, chunk_size):
122     hex_encoder = getencoder('hex')
123
124     hash = get_hash(message, chunk_size)
125     result = hex_encoder(hash)[0].decode("ascii")
126
127     return result
128
129
130 def main():
131     print('Enter the message: ', end='')
132     message = str(input())
133     msg_utf_encoded = message.encode('utf-8')
134     chunk_size = 256
135     result = hex_hash(msg_utf_encoded, chunk_size)
136
137     print('Message: {0}\n' 'Hash sum: {1}'.format(message, result))
138
139
140 if __name__ == '__main__':
141     main()

```

Пример выполнения

```

Enter the message: alex
Message: alex
Hash sum: 957dfb1f3e8fa7adc22e2b1c02c68683b39fa5050fb667948a7a2f6adfbf5a87

Process finished with exit code 0

```