

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ
Кафедра информатики

Факультет: ИНО
Специальность: ИиТП

Контрольная работа № 2
по дисциплине
“Методы защиты информации”
“Алгоритм цифровой подписи ГОСТ 3410”

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

Условие

Контрольная работа №2

Указания по выбору варианта

Рабочей программой дисциплины «Методы защиты информации» предусмотрено выполнение двух контрольных работ. Контрольная работа № 2 подразумевает изучение и программную реализацию (на языке высокого уровня) алгоритма стандарта цифровой подписи ГОСТ 3410. В качестве отчета по контрольной работе высылается листинг программной реализации, представлена в виде теста и исполняемый файл. В контрольной работе № 2 используется **один вариант** (для всех номеров зачетных книжек).

Теоретическая часть

1. Изучить алгоритм цифровой подписи ГОСТ 3410.
2. Создать и протестировать алгоритм цифровой подписи ГОСТ 3410 на языке высокого уровня.

Ссылка на гит-хаб: <https://github.com/ElephantT/MZI/tree/main/KR2>

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.

```

main.py
1  """
2  Процесс формирования ЭЦП выполняется по следующему алгоритму:
3  Вычислить хеш сообщения M: H=h(M);
4  Вычислить целое число  $\alpha$ , двоичным представлением которого является H;
5  Определить  $e = \alpha \bmod q$ , если  $e=0$ , задать  $e=1$ ;
6  Сгенерировать случайное число k, удовлетворяющее условию  $0 < k < q$ ;
7  Вычислить точку эллиптической кривой  $C=k \cdot P$ ;
8  Определить  $r = xC \bmod q$ , где xC – x-координата точки C. Если  $r=0$ , то вернуться к шагу 4;
9  Вычислить значение  $s = (rd+ke) \bmod q$ . Если  $s=0$ , то вернуться к шагу 4;
10 Вернуть значение  $r||s$  в качестве цифровой подписи.
11 """
12
13 from os import urandom
14 from hashlib import sha1
15 from codecs import getdecoder
16 from codecs import getencoder
17
18 from config import SIZE
19
20
21 def hex_encode(data):
22     hex_encoder = getencoder('hex')
23     return hex_encoder(data)[0].decode('ascii')
24
25
26 def hex_decode(data):
27     hex_decoder = getdecoder('hex')
28     return hex_decoder(data)[0]
29
30
31 def modinvert(a, n):
32     if a < 0:
33         return n - modinvert(-a, n)
34
35     t, new_t = 0, 1
36     r, new_r = n, a
37     while new_r != 0:
38         quotient = r // new_r
39         t, new_t = new_t, t - quotient * new_t
40         r, new_r = new_r, r - quotient * new_r
41     if r > 1:
42         return -1
43     if t < 0:
44         t = t + n
45     return t
46
47
48 def bytes2long(raw):
49     return int(hex_encode(raw), 16)
50
51
52 def long2bytes(n, size=SIZE):
53     res = hex(int(n))[2:].rstrip("L")
54
55     if len(res) % 2 != 0:
56         res = "0" + res
57
58     s = hex_decode(res)
59
60     if len(s) != size:
61         s = (size - len(s)) * b"\x00" + s
62

```

```

60     if len(s) != size:
61         s = (size - len(s)) * b"\x00" + s
62
63     return s
64
65
66 class GOST3410Curve(object):
67     def __init__(self, p, q, a, b, x, y, e=None, d=None):
68         self.p = p
69         self.q = q
70         self.a = a
71         self.b = b
72         self.x = x
73         self.y = y
74         self.e = e
75         self.d = d
76
77         r1 = self.y * self.y % self.p
78         r2 = ((self.x * self.x + self.a) * self.x + self.b) % self.p
79
80         if r1 != self.pos(r2):
81             raise ValueError("Invalid parameters")
82
83         self._st = None
84
85     def pos(self, v):
86         if v < 0:
87             return v + self.p
88         return v
89
90     def _add(self, p1x, p1y, p2x, p2y):
91         if p1x == p2x and p1y == p2y:
92             t = ((3 * p1x * p1x + self.a) * modinvert(2 * p1y, self.p)) % self.p
93         else:
94             tx = self.pos(p2x - p1x) % self.p
95             ty = self.pos(p2y - p1y) % self.p
96             t = (ty * modinvert(tx, self.p)) % self.p
97
98             tx = self.pos(t * t - p1x - p2x) % self.p
99             ty = self.pos(t * (p1x - tx) - p1y) % self.p
100
101         return tx, ty
102
103     def exp(self, degree, x=None, y=None):
104         x = x or self.x
105         y = y or self.y
106         tx = x
107         ty = y
108         if degree == 0:
109             raise ValueError("Bad degree value")
110         degree -= 1
111         while degree != 0:
112             if degree & 1 == 1:
113                 tx, ty = self._add(tx, ty, x, y)
114             degree = degree >> 1
115             x, y = self._add(x, y, x, y)
116         return tx, ty

```

```

117
118     def st(self):
119         if self.e is None or self.d is None:
120             raise ValueError("non twisted Edwards curve")
121         if self._st is not None:
122             return self._st
123         self._st = (
124             self.pos(self.e - self.d) * modinvert(4, self.p) % self.p,
125             (self.e + self.d) * modinvert(6, self.p) % self.p,
126         )
127         return self._st
128
129
130     # Generates public key from the private one
131     def public_key(curve, prv):
132         return curve.exp(prv)
133
134
135     # Calculates signature for provided digest
136     def sign(curve, prv, digest):
137         size = SIZE * 2
138         q = curve.q
139         e = 1 if (bytes2long(digest) % q == 0) else bytes2long(digest) % q
140
141         while True:
142             k = bytes2long(urandom(size)) % q
143             if k == 0:
144                 continue
145             r, _ = curve.exp(k)
146             r %= q
147             if r == 0:
148                 continue
149             d = prv * r
150             k *= e
151             s = (d + k) % q
152             if s == 0:
153                 continue
154             break
155         return long2bytes(s, size) + long2bytes(r, size)
156

```

```

157
158 # Verifies provided digest with the signature
159 def verify(curve, pub, digest, signature):
160     size = SIZE * 2
161
162     if len(signature) != size * 2:
163         raise ValueError("Invalid signature length")
164
165     q = curve.q
166     p = curve.p
167     s = bytes2long(signature[:size])
168     r = bytes2long(signature[size:])
169
170     if r <= 0 or r >= q or s <= 0 or s >= q:
171         return False
172
173     e = bytes2long(digest) % curve.q
174
175     if e == 0:
176         e = 1
177     v = modinvert(e, q)
178     z1 = s * v % q
179     z2 = q - r * v % q
180     p1x, p1y = curve.exp(z1)
181     q1x, q1y = curve.exp(z2, pub[0], pub[1])
182     lm = q1x - p1x
183
184     if lm < 0:
185         lm += p
186     lm = modinvert(lm, p)
187     z1 = q1y - p1y
188     lm = lm * z1 % p
189     lm = lm * lm % p
190     lm = lm - p1x - q1x
191     lm = lm % p
192     if lm < 0:
193         lm += p
194     lm %= q
195     return lm == r
196
197
198 def prv_unmarshal(prv):
199     return bytes2long(prv[::-1])
200
201
202 def pub_marshal(pub):
203     return (long2bytes(pub[1], SIZE) + long2bytes(pub[0], SIZE))[::-1]
204
205
206 def pub_unmarshal(pub):
207     pub = pub[::-1]
208     return bytes2long(pub[SIZE:]), bytes2long(pub[:SIZE])
209

```


