

利用公钥密码实现数字签名：PGP

网安1班 2018302080152 范圣悦

一、PGP流程简介

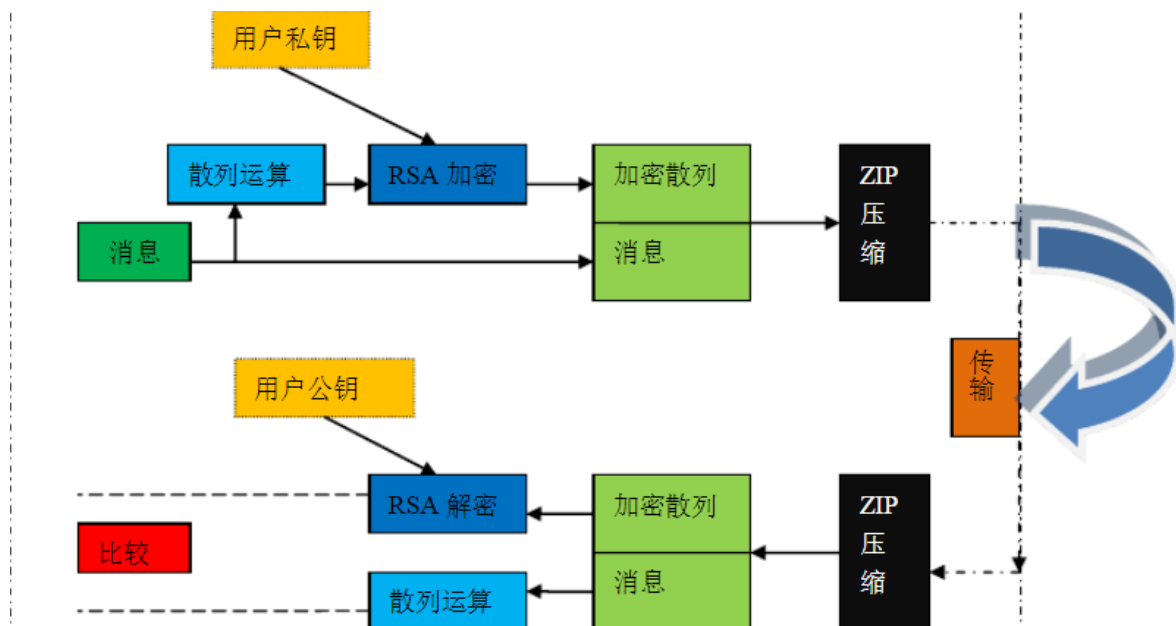
PGP是一个混合加密算法，并不是一种完全的非对称加密体系，而是由一个对称加密算法IDEA、一个非对称加密算法RSA、一个单向散列算法以及一个随机数产生器组成。

1.1 认证算法

认证过程如下：

- 发送方创建消息
- 发送方生成消息的160位散列码
- 用发送方私钥散列进行RSA加密，加到消息上
- 传输数据
- 接收方用发送方的公钥对加密部分进行RSA解密
- 接收方将剩余数据生成160位散列码
- 生成散列与解密散列进行比较，如果匹配，则认证成功

其流程图如下所示：

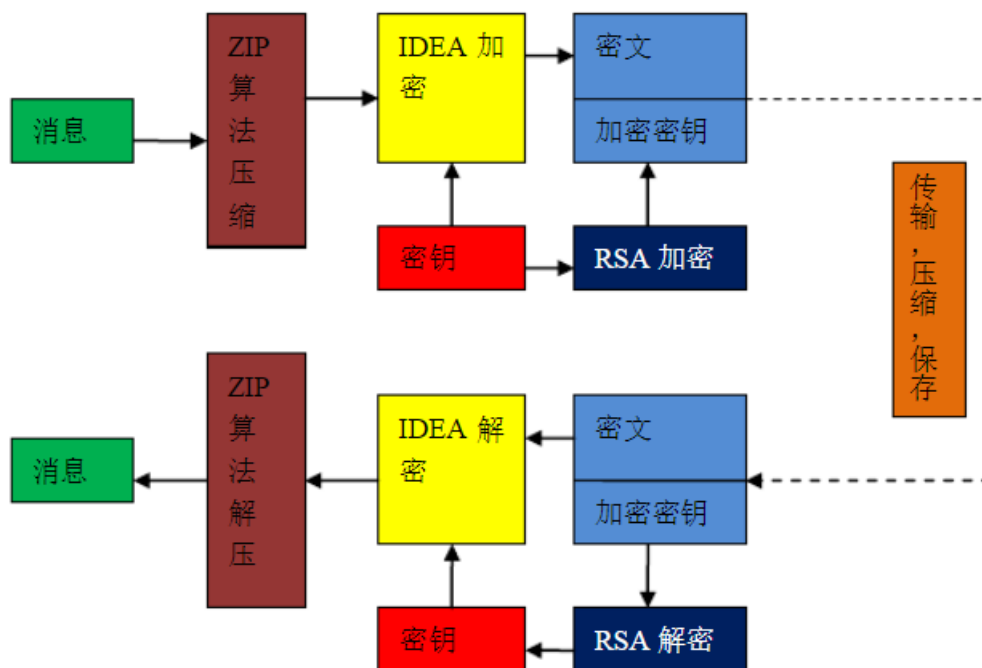


1.2 加密解密算法

加密解密过程如下：

- 发送方创建消息
- 生成128位随机码作为会话密钥（IDEA密钥）
- 用IDEA密钥对消息进行IDEA加密，得密文
- 用接收方公钥对IDEA密钥进行RSA加密，加到密文上
- 传输数据
- 接收方用私钥解密出IDEA密钥
- 用IDEA密钥解密剩余数据获得消息

其流程图如下：



1.3 保密与认证

同时使用保密和认证时，发送方用自己的私钥加密消息，然后通过PGP保密算法，用接收方公钥加密IDEA密钥；接收方接到密文通过自己的私钥，解密IDEA密钥，再解密消息，接收方用发送方的公钥解密消息后，剩余码通过散列运算再与解密消息比较。

二、核心代码实现

2.1 文件MD5散列计算算法

该部分实现md5散列计算算法，主要有以下几步：

第一步、填充：如果输入信息的长度(bit)对512求余的结果不等于448，就需要填充使得对512求余的结果等于448。填充的方法是填充一个1和n个0。填充完后，信息的长度就为 $N*512+448$ (bit)；

第二步、记录信息长度：用64位来存储填充前信息长度。这64位加在第一步结果的后面，这样信息长度就变为 $N*512+448+64=(N+1)*512$ 位。

第三步、装入标准的幻数（四个整数）：标准的幻数（物理顺序）是（A=(01234567) 16，B=(89ABCDEF) 16，C=(FEDCBA98) 16，D=(76543210) 16）。如果在程序中定义应该是

(A=0X67452301L, B=0XEFCDA89L, C=0X98BADCFEL, D=0X10325476L)。有点晕哈，其实想一想就明白了。

第四步、四轮循环运算：循环的次数是分组的个数（N+1）

```
import binascii
```

```
SV = [0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee, 0xf57c0faf,  
      0x4787c62a, 0xa8304613, 0xfd469501, 0x698098d8, 0x8b44f7af,  
      0xfffff5bb1, 0x895cd7be, 0x6b901122, 0xfd987193, 0xa679438e,  
      0x49b40821, 0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa,  
      0xd62f105d, 0x2441453, 0xd8a1e681, 0xe7d3fbc8, 0x21e1cde6,  
      0xc33707d6, 0xf4d50d87, 0x455a14ed, 0xa9e3e905, 0xfcefa3f8,
```

```
0x676f02d9, 0x8d2a4c8a, 0xfffa3942, 0x8771f681, 0x6d9d6122,  
0xfde5380c, 0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbee5fb70,  
0x289b7ec6, 0xeaad127fa, 0xd4ef3085, 0x4881d05, 0xd9d4d039,  
0xe6db99e5, 0x1fa27cf8, 0xc4ac5665, 0xf4292244, 0x432aff97,  
0xab9423a7, 0xfc93a039, 0x655b59c3, 0x8f0ccc92, 0xffeff47d,  
0x85845dd1, 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1,  
0xf7537e82, 0xbd3af235, 0xad7d2bb, 0xeb86d391]
```

```
def leftCircularShift(k, bits):  
    bits = bits % 32  
    k = k % (2 ** 32)  
    upper = (k << bits) % (2 ** 32)  
    result = upper | (k >> (32 - (bits)))  
    return (result)  
  
def blockDivide(block, chunks):  
    result = []  
    size = len(block) // chunks  
    for i in range(0, chunks):  
        result.append(int.from_bytes(block[i * size:(i + 1) * size], byteorder="little"))  
    return (result)  
  
def F(X, Y, Z):  
    return ((X & Y) | ((~X) & Z))  
  
def G(X, Y, Z):  
    return ((X & Z) | (Y & (~Z)))  
  
def H(X, Y, Z):  
    return (X ^ Y ^ Z)  
  
def I(X, Y, Z):  
    return (Y ^ (X | (~Z)))  
  
def FF(a, b, c, d, M, s, t):  
    result = b + leftCircularShift((a + F(b, c, d) + M + t), s)  
    return (result)  
  
def GG(a, b, c, d, M, s, t):  
    result = b + leftCircularShift((a + G(b, c, d) + M + t), s)  
    return (result)  
  
def HH(a, b, c, d, M, s, t):  
    result = b + leftCircularShift((a + H(b, c, d) + M + t), s)  
    return (result)  
  
def II(a, b, c, d, M, s, t):  
    result = b + leftCircularShift((a + I(b, c, d) + M + t), s)  
    return (result)  
  
def fmt8(num):  
    bighex = "{0:08x}".format(num)  
    binver = binascii.unhexlify(bighex)  
    result = "{0:08x}".format(int.from_bytes(binver, byteorder='little'))  
    return (result)  
  
def bitlen(bitstring):  
    return (len(bitstring) * 8)  
  
def md5sum(msg):  
    # First, we pad the message  
    msgLen = bitlen(msg) % (2 ** 64)  
    msg = msg + b'\x80'  
    zeroPad = (448 - (msgLen + 8) % 512) % 512  
    zeroPad //= 8  
    msg = msg + b'\x00' * zeroPad + msgLen.to_bytes(8, byteorder='little')  
    msgLen = bitlen(msg)  
    iterations = msgLen // 512  
    # chaining variables
```

```

A = 0x67452301
B = 0xefcdab89
C = 0x98badcfe
D = 0x10325476
# main loop
for i in range(0, iterations):
    a = A
    b = B
    c = C
    d = D
    block = msg[i * 64:(i + 1) * 64]
    M = blockDivide(block, 16)
    # Rounds
    a = FF(a, b, c, d, M[0], 7, SV[0])
    d = FF(d, a, b, c, M[1], 12, SV[1])
    c = FF(c, d, a, b, M[2], 17, SV[2])
    b = FF(b, c, d, a, M[3], 22, SV[3])
    a = FF(a, b, c, d, M[4], 7, SV[4])
    d = FF(d, a, b, c, M[5], 12, SV[5])
    c = FF(c, d, a, b, M[6], 17, SV[6])
    b = FF(b, c, d, a, M[7], 22, SV[7])
    a = FF(a, b, c, d, M[8], 7, SV[8])
    d = FF(d, a, b, c, M[9], 12, SV[9])
    c = FF(c, d, a, b, M[10], 17, SV[10])
    b = FF(b, c, d, a, M[11], 22, SV[11])
    a = FF(a, b, c, d, M[12], 7, SV[12])
    d = FF(d, a, b, c, M[13], 12, SV[13])
    c = FF(c, d, a, b, M[14], 17, SV[14])
    b = FF(b, c, d, a, M[15], 22, SV[15])
    a = GG(a, b, c, d, M[1], 5, SV[16])
    d = GG(d, a, b, c, M[6], 9, SV[17])
    c = GG(c, d, a, b, M[11], 14, SV[18])
    b = GG(b, c, d, a, M[0], 20, SV[19])
    a = GG(a, b, c, d, M[5], 5, SV[20])
    d = GG(d, a, b, c, M[10], 9, SV[21])
    c = GG(c, d, a, b, M[15], 14, SV[22])
    b = GG(b, c, d, a, M[4], 20, SV[23])
    a = GG(a, b, c, d, M[9], 5, SV[24])
    d = GG(d, a, b, c, M[14], 9, SV[25])
    c = GG(c, d, a, b, M[3], 14, SV[26])
    b = GG(b, c, d, a, M[8], 20, SV[27])
    a = GG(a, b, c, d, M[13], 5, SV[28])
    d = GG(d, a, b, c, M[2], 9, SV[29])
    c = GG(c, d, a, b, M[7], 14, SV[30])
    b = GG(b, c, d, a, M[12], 20, SV[31])
    a = HH(a, b, c, d, M[5], 4, SV[32])
    d = HH(d, a, b, c, M[8], 11, SV[33])
    c = HH(c, d, a, b, M[11], 16, SV[34])
    b = HH(b, c, d, a, M[14], 23, SV[35])
    a = HH(a, b, c, d, M[1], 4, SV[36])
    d = HH(d, a, b, c, M[4], 11, SV[37])
    c = HH(c, d, a, b, M[7], 16, SV[38])
    b = HH(b, c, d, a, M[10], 23, SV[39])
    a = HH(a, b, c, d, M[13], 4, SV[40])
    d = HH(d, a, b, c, M[0], 11, SV[41])
    c = HH(c, d, a, b, M[3], 16, SV[42])
    b = HH(b, c, d, a, M[6], 23, SV[43])
    a = HH(a, b, c, d, M[9], 4, SV[44])
    d = HH(d, a, b, c, M[12], 11, SV[45])
    c = HH(c, d, a, b, M[15], 16, SV[46])
    b = HH(b, c, d, a, M[2], 23, SV[47])
    a = II(a, b, c, d, M[0], 6, SV[48])
    d = II(d, a, b, c, M[7], 10, SV[49])
    c = II(c, d, a, b, M[14], 15, SV[50])
    b = II(b, c, d, a, M[5], 21, SV[51])
    a = II(a, b, c, d, M[12], 6, SV[52])
    d = II(d, a, b, c, M[3], 10, SV[53])
    c = II(c, d, a, b, M[10], 15, SV[54])
    b = II(b, c, d, a, M[1], 21, SV[55])

```

```

a = II(a, b, c, d, M[8], 6, SV[56])
d = II(d, a, b, c, M[15], 10, SV[57])
c = II(c, d, a, b, M[6], 15, SV[58])
b = II(b, c, d, a, M[13], 21, SV[59])
a = II(a, b, c, d, M[4], 6, SV[60])
d = II(d, a, b, c, M[11], 10, SV[61])
c = II(c, d, a, b, M[2], 15, SV[62])
b = II(b, c, d, a, M[9], 21, SV[63])
A = (A + a) % (2 ** 32)
B = (B + b) % (2 ** 32)
C = (C + c) % (2 ** 32)
D = (D + d) % (2 ** 32)
result = fmt8(A) + fmt8(B) + fmt8(C) + fmt8(D)
return (result)

```

2.2 生成512位质数算法

该部分主要分为两步：

- 寻找一个位数为512的随机数

利用`random.randrange()`函数，得到一个位于 $2^{(512-1)}$ 和 2^{512} 之间的随机数

- 判断该随机数是否为小素数的倍数

创建1000以内的所有小素数的列表,可以大幅加快速度；如果大数是这些小素数的倍数,那么就是合数,返回false；否则调用第三步算法

- 判断该随机数是否为质数

利用质数判断算法`rabin_miller`，该算法比第二步算法慢，但更为准确。

```

import random

def rabin_miller(num):
    s = num - 1
    t = 0
    while s % 2 == 0:
        s = s // 2
        t += 1
    for trials in range(5):
        a = random.randrange(2, num - 1)
        v = pow(a, s, num)
        if v != 1:
            i = 0
            while v != (num - 1):
                if i == t - 1:
                    return False
                else:
                    i = i + 1
                    v = (v ** 2) % num
            return True

def is_prime(num):
    # 排除0,1和负数
    if num < 2:
        return False
    # 创建小素数的列表,可以大幅加快速度
    # 如果是小素数,那么直接返回true

```

```

small_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277,
281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397,
401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509,
521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641,
643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761,
769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887,
907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997]

if num in small_primes:
    return True

# 如果大数是这些小素数的倍数,那么就是合数,返回false
for prime in small_primes:
    if num % prime == 0:
        return False

# 如果这样没有分辨出来,就一定是大整数,那么就调用rabin算法
return rabin_miller(num)

# 得到大整数,默认位数为512
def get_prime(key_size=512):
    while True:
        num = random.randrange(2**(key_size-1), 2**key_size)
        if is_prime(num):
            return num

```

2.3 检验两个数是否互质算法

该部分通过递归，用欧几里得算法判断两数是否互质。

```

def gcd(a, b):
    if a < b:
        return gcd(b, a)
    while a % b != 0:
        temp = b
        b = a % b
        a = temp
    return b

def test(a, b, c):
    d = (a-1) * (b-1)
    if gcd(d, c) == 1:
        # 互质
        return 1
    else:
        # 不互质
        return 0

```

2.4 RSA加密和解密算法

RSA算法中包括欧几里得算法、 $(base \wedge exponent) \bmod n$ 的算法、生成公钥和密钥算法。

- 生成公钥和密钥

get_key()函数的传入参数为p、q、e，计算得到phi，求出d，得到公钥(n, e)和密钥(n, d)

- 加解密

encrypt()函数和decrypt()函数传入参数分别为公钥、明文和私钥、密文。通过超大整数超大次幂然后对超大的整数取模函数exp_mode()进行加解密的运算。

```

# 求两个数字的最大公约数（欧几里得算法）
def gcd(a, b):
    if b == 0:

```

```

        return a
    else:
        return gcd(b, a % b)

'''
扩展欧几里的算法
计算  $ax + by = 1$  中的  $x$  与  $y$  的整数解 ( $a$  与  $b$  互质)
'''
def ext_gcd(a, b):
    if b == 0:
        x1 = 1
        y1 = 0
        x = x1
        y = y1
        r = a
        return r, x, y
    else:
        r, x1, y1 = ext_gcd(b, a % b)
        x = y1
        y = x1 - a // b * y1
        return r, x, y

'''
超大整数超大次幂然后对超大的整数取模
(base ^ exponent) mod n
'''

def exp_mode(base, exponent, n):
    bin_array = bin(exponent)[2:][::-1]
    r = len(bin_array)
    base_array = []

    pre_base = base
    base_array.append(pre_base)

    for _ in range(r - 1):
        next_base = (pre_base * pre_base) % n
        base_array.append(next_base)
        pre_base = next_base

    a_w_b = __multi(base_array, bin_array, n)
    return a_w_b % n

def __multi(array, bin_array, n):
    result = 1
    for index in range(len(array)):
        a = array[index]
        if not int(bin_array[index]):
            continue
        result *= a
        result = result % n # 加快连乘的速度
    return result

# 生成公钥私钥, p、q为两个超大质数
def get_key(p, q, e):
    n = p * q
    fy = (p - 1) * (q - 1) # 计算与n互质的整数个数 欧拉函数
    # generate d
    a = e
    b = fy
    r, x, y = ext_gcd(a, b)
    # 计算出的x不能是负数, 如果是负数, 说明p、q、e选取失败, 不过可以把x加上fy, 使x为正数, 才能计算。
    if x < 0:
        x = x + fy
    d = x
    # 返回:      公钥      私钥
    return (n, e), (n, d)

# 加密 m是被加密的信息 加密成为c

```

```
def encrypt(m, pubkey):
    n = pubkey[0]
    e = pubkey[1]

    c = exp_mode(m, e, n)
    return c

# 解密 c是密文，解密为明文m
def decrypt(c, selfkey):
    n = selfkey[0]
    d = selfkey[1]

    m = exp_mode(c, d, n)
    return m
```

2.5 IDEA算法

IDEA加密算法用了52个子密钥（8轮中的每一轮需要6个，其他4个用于输出变换）。首先，将128-位密钥分成8个16-位子密钥。这些事算法的第一批8个子密钥（第一轮6个，第二轮头两个）。然后，密钥向左环移动x位后再分成8个子密钥。开始4个用在第二轮，后面四个用在第三轮。密钥再次向左环移动25位，产生另外8个子密钥，如此进行指导算法结束。

具体是：IDEA总共进行8轮迭代操作，每轮需要6个子密钥，另外还需要4个额外子密钥，所以总共需要52个子密钥，这52个子密钥都是从128位密钥中扩展出来的。

首先把输入的key分成8个16位的子密钥，1-6号子密钥供第一轮加密使用，7-8号子密钥供第二轮使用，然后把这128位密钥循环左移25位，这样key=k26k27k28k29.....k24k25。

把新生成的key在分成8个16位的子密钥，1-4号子密钥供第二轮使用，5-8号子密钥供第三轮加密使用。到此，已经得到了16个子密钥，如此继续，当循环左移了5次之后，已经生成了48个子密钥，还有四个额外的子密钥需要生成，再次把key循环左移25位，选取划分出来的8个16位子密钥的前四个作为那四个的额外加密密钥。至此，供加密使用的52个子密钥生成完毕。

输入的64-位数据分组被分成4个16-位子分组：x1,x2,x3和x4。这4个子分组成为算法的第一轮的输入，总共有8轮。在每一轮中，这4个子分组相互相异或，相乘，相加，且与6个16-位子密钥相异或，相乘，相加。在轮与轮间，第二个和第三个子分组交换。最后输出变换中4个子分组与4个子密钥进行运算。

```
import gmpy2

def _mul(x, y):
    assert 0 <= x <= 0xFFFF
    assert 0 <= y <= 0xFFFF
    if x == 0:
        x = 0x10000
    if y == 0:
        y = 0x10000
    r = (x * y) % 0x10001
    if r == 0x10000:
        r = 0
    assert 0 <= r <= 0xFFFF
    return r

def _KA_layer(x1, x2, x3, x4, round_keys):
    assert 0 <= x1 <= 0xFFFF
    assert 0 <= x2 <= 0xFFFF
    assert 0 <= x3 <= 0xFFFF
    assert 0 <= x4 <= 0xFFFF
    z1, z2, z3, z4 = round_keys[0:4]
    assert 0 <= z1 <= 0xFFFF
    assert 0 <= z2 <= 0xFFFF
    assert 0 <= z3 <= 0xFFFF
    assert 0 <= z4 <= 0xFFFF
    y1 = _mul(x1, z1)
    y2 = (x2 + z2) % 0x10000
```



```

y3 = (x3 + z3) % 0x10000
y4 = _mul(x4, z4)
return y1, y2, y3, y4

def _MA_layer(y1, y2, y3, y4, round_keys):
    assert 0 <= y1 <= 0xFFFF
    assert 0 <= y2 <= 0xFFFF
    assert 0 <= y3 <= 0xFFFF
    assert 0 <= y4 <= 0xFFFF
    z5, z6 = round_keys[4:6]
    assert 0 <= z5 <= 0xFFFF
    assert 0 <= z6 <= 0xFFFF
    p = y1 ^ y3
    q = y2 ^ y4
    s = _mul(p, z5)
    t = _mul((q + s) % 0x10000, z6)
    u = (s + t) % 0x10000
    x1 = y1 ^ t
    x2 = y2 ^ u
    x3 = y3 ^ t
    x4 = y4 ^ u
    return x1, x2, x3, x4

class IDEA:
    def __init__(self, key):
        self._expand_key = []
        self._encrypt_key = None
        self._decrypt_key = None
        self.expand_key(key)
        self.get_encrypt_key()
        self.get_decrypt_key()

    def expand_key(self, key):
        assert 0 <= key < (1 << 128)
        modulus = 1 << 128
        for i in range(6 * 8 + 4):
            self._expand_key.append((key >> (112 - 16 * (i % 8))) % 0x10000)
            if i % 8 == 7:
                key = ((key << 25) | (key >> 103)) % modulus
        return self._expand_key

    def get_encrypt_key(self):
        keys = []
        for i in range(9):
            round_keys = self._expand_key[6 * i:6 * (i + 1)]
            keys.append(tuple(round_keys))
        self._encrypt_key = tuple(keys)

    def get_decrypt_key(self):
        keys = [0] * 52
        for i in range(9):
            if i == 0:
                for j in range(6):
                    if j == 0 or j == 3:
                        if self._encrypt_key[8 - i][j] == 0:
                            keys[j] = 0
                    else:
                        keys[j] = gmpy2.invert(self._encrypt_key[8 - i][j],
                                              65537)
            elif j == 1 or j == 2:
                keys[j] = (65536 - self._encrypt_key[8 - i][j]) % 65536
            else:
                keys[j] = self._encrypt_key[7 - i][j]
        elif i < 8:
            for j in range(6):
                if j == 0 or j == 3:
                    if self._encrypt_key[8 - i][j] == 0:
                        keys[i * 6 + j] = 0
                else:

```

```

        keys[i * 6 + j] = gmpy2.invert(
            self._encrypt_key[8 - i][j], 65537)
    elif j == 1 or j == 2:
        keys[i * 6 + 3 -
            j] = (65536 - self._encrypt_key[8 - i][j]) % 65536
    else:
        keys[i * 6 + j] = self._encrypt_key[7 - i][j]
    else:
        for j in range(4):
            if j == 0 or j == 3:
                if self._encrypt_key[8 - i][j] == 0:
                    keys[i * 6 + j] = 0
            else:
                keys[i * 6 + j] = gmpy2.invert(
                    self._encrypt_key[8 - i][j], 65537)
        else:
            keys[i * 6 +
                j] = (65536 - self._encrypt_key[8 - i][j]) % 65536

tmp = []
for i in range(9):
    round_keys = keys[6 * i:6 * (i + 1)]
    tmp.append(tuple(round_keys))
self._decrypt_key = tuple(tmp)

```

```

def enc_dec(self, plaintext, flag):
    assert 0 <= plaintext < (1 << 64)
    x1 = (plaintext >> 48) & 0xFFFF
    x2 = (plaintext >> 32) & 0xFFFF
    x3 = (plaintext >> 16) & 0xFFFF
    x4 = plaintext & 0xFFFF
    if flag == 0:
        key = self._encrypt_key
    else:
        key = self._decrypt_key
    for i in range(8):
        round_keys = key[i]
        y1, y2, y3, y4 = _KA_layer(x1, x2, x3, x4, round_keys)
        x1, x2, x3, x4 = _MA_layer(y1, y2, y3, y4, round_keys)
        x2, x3 = x3, x2
        # Note: The words x2 and x3 are not permuted in the last round
        # So here we use x1, x3, x2, x4 as input instead of x1, x2, x3, x4
        # in order to cancel the last permutation x2, x3 = x3, x2
        y1, y2, y3, y4 = _KA_layer(x1, x3, x2, x4, key[8])

    ciphertext = (y1 << 48) | (y2 << 32) | (y3 << 16) | y4
    return ciphertext

```

IDEA加密

```

def IDEA_en(key, M):
    my_IDEA = IDEA(key)
    original_len = len(M)
    if len(M) % 8 != 0:
        #PADDING
        #填充后的长度是64bits的整数倍 即8bytes的整数倍
        M = M + bytes([1])
        pad_zero_len = 0
        while((pad_zero_len + original_len + 2) % 8 != 0):
            pad_zero_len += 1
        M = M + bytes(pad_zero_len) # 填充这么多个0
        M = M + bytes([pad_zero_len]) # 填充一个数: original_len
        # print(M)
    LEN = int(len(M) / 8)
    Cipher = bytes()
    for i in range(LEN):
        plain = M[i*8: i*8+8]
        plain = int.from_bytes(plain, byteorder='little', signed=False)
        #密文 10进制形式
        encrypted = my_IDEA.enc_dec(plain, 0)
        Cipher = Cipher + int(encrypted).to_bytes(8, byteorder='little', signed=False)

```

```

    # print(Cipher)
    return Cipher

# IDEA解密
def IDEA_de(Cipher, key):
    my_IDEA = IDEA(key)
    Decrypted = bytes()
    LEN = int(len(Cipher) / 8)
    for i in range(LEN):
        cipher = Cipher[i*8:i*8+8]
        cipher = int.from_bytes(cipher, byteorder='little', signed=False)
        decrypted = my_IDEA.enc_dec(cipher, 1)
        #print(int(decrypted).to_bytes(8, byteorder='little', signed=False))
        Decrypted = Decrypted + int(decrypted).to_bytes(8, byteorder='little', signed=False)
    pad_zero_len = Decrypted[-1]
    original_len = len(Cipher) - 2 - pad_zero_len
    Decrypted = Decrypted[:original_len]
    # print(Decrypted)
    return Decrypted

```

2.6 Base64编码解码算法

```

# base 字符集
ascii_lowercase = 'abcdefghijklmnopqrstuvwxyz'
ascii_uppercase = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
digits = '0123456789'
base64_charset = ascii_uppercase + ascii_lowercase + digits + '+/'

def encode(origin_bytes):
    """
    将bytes类型编码为base64
    :param origin_bytes: 需要编码的bytes
    :return: base64字符串
    """
    # 将每一位bytes转换为二进制字符串
    base64_bytes = ['{:0>8}'.format(str(bin(b)).replace('0b', '')) for b in origin_bytes]

    resp = ''
    nums = len(base64_bytes) // 3
    remain = len(base64_bytes) % 3

    integral_part = base64_bytes[0:3 * nums]
    while integral_part:
        # 取三个字节，以每6比特，转换为4个整数
        tmp_unit = ''.join(integral_part[0:3])
        tmp_unit = [int(tmp_unit[x: x + 6], 2) for x in [0, 6, 12, 18]]
        # 取对应base64字符
        resp += ''.join([base64_charset[i] for i in tmp_unit])
        integral_part = integral_part[3:]

    if remain:
        # 补齐三个字节，每个字节补充8个0
        remain_part = ''.join(base64_bytes[3 * nums:]) + (3 - remain) * '0' * 8
        # 取三个字节，以每6比特，转换为4个整数
        # 剩余1字节可构造2个base64字符，补充==; 剩余2字节可构造3个base64字符，补充=
        tmp_unit = [int(remain_part[x: x + 6], 2) for x in [0, 6, 12, 18]][:(remain + 1)]
        resp += ''.join([base64_charset[i] for i in tmp_unit]) + (3 - remain) * '='

    return resp

def decode(base64_str):
    """
    解码base64字符串
    :param base64_str: base64字符串
    :return: 解码后的bytes
    """

```

```

"""

# 对每一个base64字符取下标索引，并转换为6为二进制字符串
base64_bytes = ['{:0>6}'.format(str(bin(base64_charset.index(s))).replace('0b', '')) for s
in base64_str if
                s != '=']
resp = bytearray()
nums = len(base64_bytes) // 4
remain = len(base64_bytes) % 4
integral_part = base64_bytes[0:4 * nums]

while integral_part:
    # 取4个6位base64字符，作为3个字节
    tmp_unit = ''.join(integral_part[0:4])
    tmp_unit = [int(tmp_unit[x: x + 8], 2) for x in [0, 8, 16]]
    for i in tmp_unit:
        resp.append(i)
    integral_part = integral_part[4:]

if remain:
    remain_part = ''.join(base64_bytes[nums * 4:])
    tmp_unit = [int(remain_part[i * 8:(i + 1) * 8], 2) for i in range(remain - 1)]
    for i in tmp_unit:
        resp.append(i)

return resp

```

2.7 加密部分流程算法

加密整个流程如下：

- 1.对M进行MD5散列计算，得到MD5(M)
- 2.利用RSA加密MD5(M)，得到S
- 3.利用ZIP压缩<M,S>
- 4.利用IDEA加密压缩数据
- 5.用RSA加密IDEA的密钥k，得到RSA(k)
- 6.把IDEA加密后的压缩数据和RSA(k)拼接在一起，并转换为BASE64

```

# 读取文件
fp = open("ys168.com.txt", "rb")
msg = fp.read()
fp.close()

'''*****加密过程*****'''

'''1. 对M进行MD5散列计算，得到MD5(M)'''
md5_hash = md5.md5sum(msg)

'''2. 利用RSA加密MD5(M)，得到S'''
# 公钥私钥中用到的两个大质数p,q，都是512位；e是和(q-1)*(p-1)互质的另一个正整数
p1 = get512prime.get_prime()
q1 = get512prime.get_prime()
while p1 == q1:
    q1 = get512prime.get_prime()
for i in range(10000):
    # 若phi和e互质
    if huzhi.test(p1, q1, i):
        e1 = i
        break

# 1是发送者，2是接收者
# 生成发送者公钥私钥

```

```

pubkey_1, selfkey_1 = rsa.gen_key(p1, q1, e1)

# 把hash值:十六进制->十进制
m = int(md5_hash, 16)
# 用发送者的私钥对hash进行加密得到签名S
S = rsa.decrypt(m, selfkey_1)
# 签名S共1024位, 不足的高位补0
S = '{:01024b}'.format(S)
S = S.encode('utf-8') # 转bytes类型

'''3.利用ZIP压缩 < M, S >'''
# pass

'''4. 利用IDEA加密压缩数据'''
# 拼接<M,S>
new_msg = msg
# 把1024位二进制的S转为128位16进制
for i in range(int(len(S) / 8)):
    tmp = S[i * 8: i * 8 + 8]
    data = int(tmp, 2)
    data = bytes([data])
    new_msg = new_msg + data # 拼接, 后128位为签名S
# print(new_msg[-128:])
# 生成一个随机的128位IDEA密钥
IDEA_key = 0x2BD6459F82C5B300952C49104881FF48
# 对拼接后的数据进行IDEA加密
IDEA_MS = IDEA.IDEA_en(IDEA_key, new_msg)

'''5. 用RSA加密IDEA的密钥k, 得到RSA(k)'''
# 公钥私钥中用到的两个大质数p,q, 都是512位; e是和(q-1)*(p-1)互质的另一个正整数
p2 = get512prime.get_prime()
q2 = get512prime.get_prime()
while p2 == q2:
    q1 = get512prime.get_prime()
for i in range(10000):
    # 若phi和e互质
    if huzhi.test(p2, q2, i):
        e2 = i
        break
# 生成接收者公钥私钥
pubkey_2, selfkey_2 = rsa.gen_key(p2, q2, e2)
# 用接收者的公钥加密IDEA的密钥
RSA_k = rsa.encrypt(IDEA_key, pubkey_2)
# 高位补0 共1024位
RSA_k = '{:01024b}'.format(RSA_k)
RSA_k = RSA_k.encode('utf-8') # 转bytes类型

# 6.把IDEA加密后的压缩数据和RSA(k)拼接在一起,并转换为BASE64
data_part = IDEA_MS
for i in range(int(len(RSA_k) / 8)):
    tmp = RSA_k[i * 8: i * 8 + 8]
    data = int(str(tmp, encoding='utf8'), 2) # bytes->str->int
    data = bytes([data]) # int->bytes
    data_part = data_part + data # 拼接 后128位为S
# 进行BASE 64变化
base64_C = base64.b64encode(data_part)
# print(base64_C)

```

2.8 解密部分流程算法

解密整个流程如下:

- 1.base64解码, 拆解消息部分与加密密钥部分
- 2.RSA解密IDEA密钥K(128位)
- 3.IDEA解密得到消息部分明文

4.拆解消息部分和加密散列部分

5.用发送者公钥解密数字签名部分S，得到hash1；计算消息部分的hash2。

6.对比hash1和hash2，若一致，那么签名验证成功

```
'''*****解密过程*****'''

'''1. base64解码，拆解消息部分与加密密钥部分'''
#UNZIP(C3) = <C1, C2>
C3 = base64.b64decode(base64_C)
# 消息部分
C1 = C3[:-128]
# 加密密钥部分
C2 = C3[-128:]

'''2. RSA解密IDEA密钥K(128位)'''
C2_bin = ''.encode('utf-8')
for i in range(int(len(C2))):
    tmp = C2[i]
    tmp = int(tmp) # bytes->int
    tmp = '{:08b}'.format(tmp).encode('utf-8') # int->str->bytes
    C2_bin = C2_bin + tmp
C2 = int(str(C2_bin.decode('utf-8')), 2)
IDEA_key = rsa.decrypt(C2, selfkey_2)

'''3. IDEA解密消息部分明文M2'''
# 用解得的密钥K解密IDEA，得到明文
M2 = IDEA.IDEA_de(C1, IDEA_key)

'''4. 拆解M与S'''
# 消息部分
Message = M2[:-128]
# 签名部分
S = M2[-128:]
S_bin = ''.encode('utf-8')
for i in range(int(len(S))):
    tmp = S[i]
    tmp = int(tmp) # bytes->int
    tmp = '{:08b}'.format(tmp).encode('utf-8') # int->str->bytes
    S_bin = S_bin + tmp
S = int(str(S_bin.decode('utf-8')), 2)

'''5. 验证签名'''
# 用发送者公钥解密数字签名部分S，得到hash
M1 = rsa.encrypt(S, pubkey_1)
M1 = hex(M1)[2:]
print('解密签名得到的hash:')
print(M1)

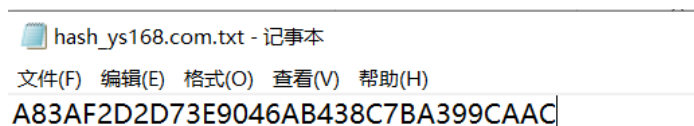
# 计算消息部分的hash，与解密S得到的hash进行对比，若一致，则解密成功
md5_Message = md5.md5sum(Message)
print('计算消息部分的hash:')
print(md5_Message)
if md5_Message == M1:
    print("验签认证成功")
else:
    print("认证失败，签名错误")
```

三、加解密运行截图和验签认证

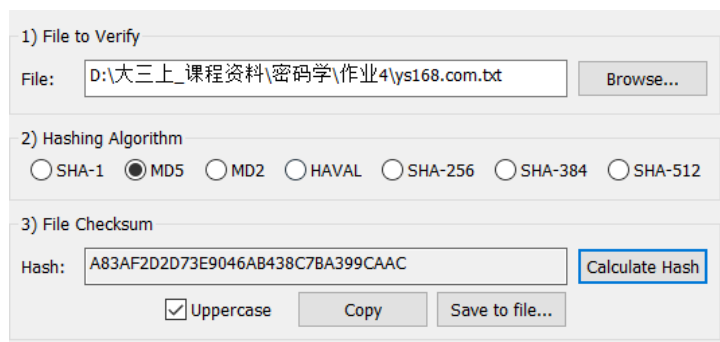
① 加密

3.1.1 对目标文件进行MD5散列计算并验证其正确性

对ys168.com.txt文件计算MD5值，并写入hash_ys168.com.txt文件，结果如下：



与HASH计算软件得到的结果对比，结果如下：



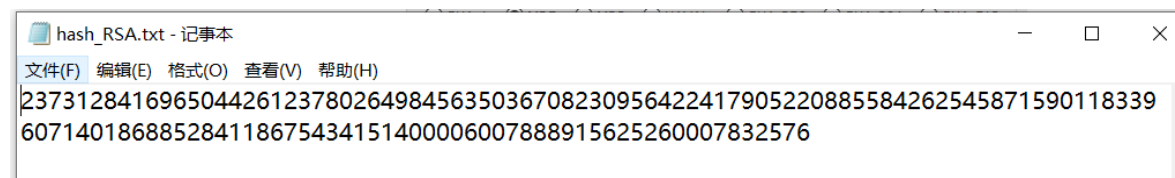
发现MD5计算正确。

3.1.2 利用RSA加密MD5(M)

生成两个512位的大质数，和与phi互质的一个正整数e，算法可以根据p、q、e得到发送者的公钥(n, e)，和私钥(n, d)

```
p1 =  
11892845164754857302192817993787160730215019683720078508868206227442930130  
72282506409181599876427341004631349534103413555781950233333807441597619886  
6323473  
  
q1 =  
11367670600867484708699649015361923450974263455941875059545547729179804359  
36189195408918604681445309265008579810704446829840084019314139571034081543  
9355097  
  
phi = (p1-1) * (q1-1)  
  
e1 = 313
```

对hash_ys168.com.txt文件，用发送者的私钥进行RSA解密，得到的数据写入hash_RSA文件，如下：



3.1.3 利用ZIP压缩拼接后的数据

压缩后的数据写入文件text.txt

3.1.4 利用IDEA加密压缩数据

对压缩后的数据text.txt进行IDEA加密，其中IDEA的128位密钥如下：

IDEA密钥：0x4AD6459F82C5B300952C49104881EF51

加密后的密文写入IDEA_text.txt文件，如下：



```
30ee9458a3d370ed8558da2e1a871731504169744e7bb7df2d684936bffb5daa9a4e3adb6b263038cec1
59440b930bb388c51e0e5688cd22e9d55672e076f27ca39e376439f7304acd1d7130cb62f6a24019ff607
cb071ac6f4b6baf580ee025f8df4d1cd48cf693c6bbc8a0305e644c4e00e32834ef797dfe4c819024cdf2a3
854cc2a562c0c73d00e87fdafd4ae7780d7fb20d4d11086d49e84098b84f3a2bbdf86e3423adaf49765a2
1e061e5c789110f6cb8a740571b04bd03d669016ba2c8f7343974d8561e47475a0d85f3f5a862e55f6a2e
8315e2ea451f7ebc5f29b6374f27d3262bd561e32ed91f0a7493adae5a36f934413d8297519e3a7f7e43cc
eb1ad0e60127bb23a3c83d389f255ecc2e85be42434f9588a0a729feefd4c49dd80e7a7a7adc5cda1119
1667ccc5754c272076ea0d4fdc16a65327e54b6c713a0a729feefd4c49dd80e7a7a7adc5cda11191667cc
c57543a38f06de5a8ca3c4f5bf2036b22c62a71aebfc7a4db3f420c3f6edcc28208cf38be54cd2c86e62daa
34c8f0f74f53e436e6e225ccc049f73d642f02f3b7dea27fdc7f18fc8f8fb0b3b3c074bac2a7ae8c82261c2c7
1aa326806f39e339222ffa5aa00737215733317466682c1911fec856f6eacffe2997e36e6e225ccc049f7c06
d4f3c8e909c73fd1255f6c08fab988a56108f29feed140f5681213734f2fe7a947560514081fce52115262f8
18842c2d3d578171c43d36606392e8d7222e30c6551d8ce0a5b75bc67d19ac71017b3d00b5ddb297
d479db14295b78cc81b045f08b876676c80c6a68e5b63e7685345da52e945eebe9d5d246ec4253cefb82
7a92e95775d3508ee550bd25c1aad9539ca30b76815ac023a08d63b2adb13c3e5f8ae8d8744cd31ef5fec
9dbd8a10ac06c3c2ef775860efd8a4e2ad5f0973cd30cc43707463b66d870af3a6749a15be1c29a864c58
ce11ebc8a257a6e97fe1994d40e0a17ed4bacd14c0efa715aab6cba9ecc4c5f1667eaf884a654fda473603
6556a055d2ec25ccad1359dc200078f94ed1f74e219514132a8c68a7aeb3a105840131957429283806b
f32ace2cde60f65cf17b8640d49a7a89fb110955bc8b79ccbcb66a09da59781ff67f6b0c7d0f2a73979d1
```

3.1.5 用RSA加密IDEA的密钥k，得到RSA(k)

生成两个512位的大质数，和与phi互质的一个正整数e，算法可以根据p、q、e得到接收者的公钥(n, e)，和私钥(n, d)

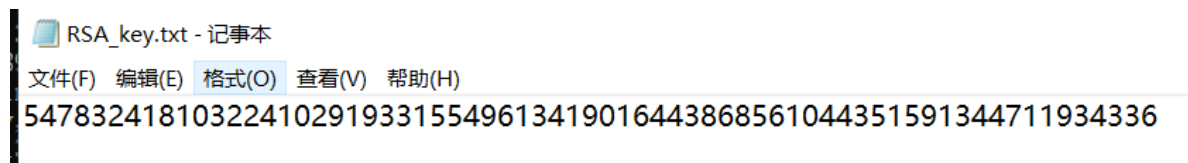
```
p2 =
96425271618482980712015451831095834412103249136728878127740311635402925559
23225605591510697538531861645360860564168061047846750456591161660397179909
428673

q2 =
12462877728415710567613768729223820618177080415134397242805949239369738252
51478705826070960131783348789137976718388286043258397953323400946019039559
2629511

phi = (p2-1) * (q2-1)

e2 = 931
```

用接收者的公钥加密IDEA的密钥K，得到RSA_key.txt，如下：



```
5478324181032241029193315549613419016443868561044351591344711934336
```

3.1.6 把IDEA加密后的消息和RSA(k)拼接在一起,并转换为BASE64

把拼接并转换为base64的数据流写入base_64.txt文件中，如下图所示：


```
base64_text.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
FOWGlz7d6g7ikbhpT9AbxPrVJk1KWnwC3u6JtmXP6eIO/CCpLI7WzmcFTY549aZpv2J/OQjX0808BFfOU8
tgifiOa01oeHx3OvQ/AAPKNVIEALeB+
+EQ7GTPFAIRbgyaPmVLuJ2MSjZglQ05CHGwExQj3+col7DrtBXq7n85Z7lyDnq0uE3uaToUHR66m/DZRz
cYqB4HMzjEd/Sh2G9zkhNexg4L7BywSOLvmRr1VS7E8IMDs70hc85cleJfKcWxgNnSKwso1WjGfDo5gV
Za
+vgOarhWo1451owiiP6aiU7SH/oiMYIbLISvJldLEbaSLvbrfUComQjWlmpxH4nQ4xVygLhhtaEde5gPDGeR
1slTgS0b+bmAacthr7VCLIns1RErXyKVLQUFVQOeMPW2I1MewX2g7y3OyU
+RO5QnoBi6mgm7DofxAVgueRW2KVZnn8j
+2WnZu2U9nuYUZbpgCmkvkWslIGQWuJVfqQ9OAJVKSyeH8NXk6SevJldLEbaSLtFR1Wm2dd1H8srFZd
Vgeg1E8P9pT6ColG5BZeroND/6zd+b5NLoqUne+5kbsxPFe0Lk8czFES0k
+zl5yBsrO3sML8Q3Qb3detkwOlGWS6Si7c3IK2btuyaRb9QsraY5ykMzH0R8mNfQrtT3Q7AYh3WzHDtLm
BjB8uBRPg0/jizCRTrP04fhCPoTyh9X8yCjf1p9MJ55KAmiySO4ECrv8EK6RvA0ukdBQ2gS0mjvESyxbQT3
h74OBCLr5XDtPhnq7DNUdaay/TyvBBT9g6xTLIDF9ZVai1jTtaOYAodeYPhrC0mPaNY1Q/FaAXrlOywxQCO
YwNdxH1IOOpTd3ssU0uLqw2vg/PwlPYupBKd6vjdlDObLCwXeAJ1K6X0SnAHan0V/ebmwEHitblgMNG
uP3b6oMZYoABGWtAe0WZnzFQklQij1X7Oa5gUiC5D9CzNCUqg
+riIv3IOV1HuSdb59iArQ1JJMI01JRwx2j668A9/dw4RdJuRvzGccLX50oM6dMc
+WBCY2yNwthsN7I2LEK0rQ6lx6gxOB2j6MAGxZTEgmsmjpm+ziejim2t/tPckwJd+hMxPq/C3uX
+M2tGbEq/HkESr9cNejLEsEGJrN+PistDPJ
+Q5Ctcqwt96H3PUnShTS6UBHQaFi84smoGi6GWRCxj0C4e461VZJ6MF
第 1 行, 第 4779 列 100% Windows (CRLF) UTF-8
```

② 解密

3.2.1 base64解码，拆解消息部分与加密密钥部分

拆解消息部分与加密密钥部分，得到消息部分：

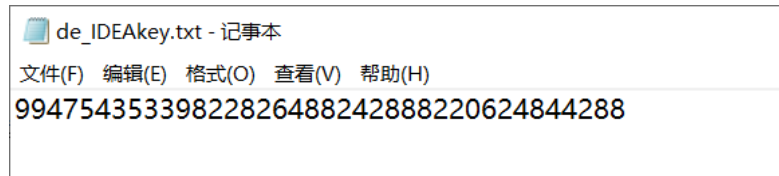
```
\x1ev-, \x8f\x84\xd0\x03\xb7\xf4k\x9f2\t\x00\xba\x87K\xef\xde:\xf7\xdcc\xcc\xcf\x1\x89b_
\xea\x96\xe4\xb5"X8\xad\xe9\xae\x0c\xc2\xc4h\xca{\x97\x96\xb6:\x9f\x853\xea\xf7\x11\xa
e\xf9\x5f\x87\x91y)\x1cm\xfc:\x1c\x03\xba\x0e*#\xb7\xda\xbf$G7G\x9\xef\xb9Z\xed\xa3\
xbd\xa5\xad\x84\xa8\x0f?
\x98\x97\x00\xae\x97kRj\x96@71|\xd9\xf6\xc9\xd2L/q\x94\xd9\xd1\x9c\xd6u"\xdf\xa9\x18
\x12MI\x1c\x1f6\x9b\xa1\xef\xd6\xaf&WK\x11\xb6\x92.\x13\x00\x1d\xb4'r\x88Q\xc9\x11[\x
e2k\x82l\xe0o\x8a\xe8\x14$\xf6\xee7\xac\xa0\xf3#k|y\xcb2\xb4\xfa8\xdau\x06\xe4\xc3\x8
1:\xd1\xf8\xa8\xb85a\xc6\xac\xbf\x92\xbc\xec\xd2\xf5\x99\xc9\x18\x96-
\xe3\xcc\x82\xf2c&\x02\xb7\x8b\x1d\x15-
\x8c\x7ff\xd1\x8d\n\x96\x97f\xd2dt\xe7\xa4\xe6` \xc3\xd1\xfe\xf9\xdd\xad\x87\xe1\x9f\x8
7\xa8.B1\x9d\xc5\xb9C\x00\xb4\xd5)E\xed\xf4ly\xaf\xf7\x92\xc52\x03\x1e\x82<\xc4X\x9f\x
10\xd8\xe3[\xac\x8e\x9a\xc1\xc5\xd5g\xd1\x80\xfe1\x9d\xc5\xb9C\x00\xb4\xd5\xba\xc8\x
87\xb9\xcd\x82\x05E\xf9#\x13\xf2>\xf1\x9bRk\x1c\xfb6Mk\xd5\x9dli\xcf\x06+\xc4\x18F\x1f
+A\xdc\x98,\xc0\xe5\xf1\xbc\xac\x0fH\xaf\x10\xd4\xc3DBM#)\x19\xd1\xaf\xe5O\x15\x15X
d\xc8Z\xc3\xe3\xf4\xae\x85Q\xeb\xaa\x8f\x1e\x1cS\xf3\xe8=\xd3w\xc9\xc7\x86\xaf\x83s\
xb7(\xfb\xdf\x99\xe6{G\xf8\x01-\xe9U?
\xf9\xa1\x971\xf1\xe5jp"\xb3\x03\xd7\x88\xd2|3\xa7\x04\xd31\x8b\xf2\xb6\x87\xcb\xac<|
=\xb0r\x8b%\xb9\x04\x9dr\x7f\x84\xe5d\xe6\xe5\xf7\x157\xae\x9e...
```

得到加密密钥部分：

```
b"}EU\x06\x83\xe5\xbeb\xdd\x16\x0c\xdfvp\xeb\x19\xac\xe1\xb5[\xb9v\x9f\xa8\xff\xaf\xe8
\x89\xc3S\x02wK\xda\xaa\xb1\xe4\xf6\xfb/\xdfc_\^xcc\x97\x18\x1d\x89` \xb5\xdfk0t\xe4\x
f8O\nf\b9+\x83\xca^\xd7\xca\xdb\x9dK\x04\x18\xaa\x00\x02\xbe\x9a|\xee\xcc\xac\xea\x
c7y\xc7?
G0\x81|\x1et\xfd%\xcc\xf7"\xd8\xbd\xab\xc7\xce\xfeS\xdej\xd1J\r\xddq\xf9\x11\x0e\`xe6\x
db \xe0\x80D\x83w\x0b\x07\xab\xcc...
```

3.2.2 RSA解密IDEA密钥K(128位)

利用接收者私钥解密“加密密钥部分”，解密出来得到IDEA的密钥K，写入文件de_IDEAkey.txt中，如下图所示：



转为16进制之后为：

```
0x4AD6459F82C5B300952C49104881EF51
```

与之前的密钥一致，揭秘正确。

3.2.3 IDEA解密消息部分得到明文

用上一步得到的IDEA密钥K解密消息部分，得到的数据流写入文件de_IDEA_text.txt中，如下图所示：



3.2.4 拆分得到加密散列S和消息部分M

上一步得到的数据流中，后128位为加密散列S，前面部分为真正的消息M，将两者分离。

得到加密散列S，即RSA加密后的MD5散列值：

```
23731284169650442612378026498456350367082309564224179052208855842625458715
9011833960714018688528411867543415140000600719373957062083164864
```

3.2.5 用发送者公钥用RSA解密加密散列S，得到实际发送过来的hash

用发送者公钥用RSA解密加密散列S，得到实际发送过来的hash，得到的hash值为：

A83AF2D2D73E9046AB438C7BA399CAAC

3.2.6 计算消息部分的hash

利用3.2.4中得到的消息部分M，计算M的hash，得到的hash值为：

A83AF2D2D73E9046AB438C7BA399CAAC

③ 验签认证

用发送者公钥用RSA解密加密散列S，得到hash为：

A83AF2D2D73E9046AB438C7BA399CAAC

计算消息部分得到的hash为：

A83AF2D2D73E9046AB438C7BA399CAAC

发现二者一致，说明签名验证成功！

截图如下：

```
解密签名得到的hash：  
a83af2d2d73e9046ab438c7ba399caac  
计算消息部分的hash：  
a83af2d2d73e9046ab438c7ba399caac  
验签认证成功  
  
Process finished with exit code 0
```

四、测速

本次PGP实现使用的文件为ys168.com.txt，文件大小为12,191,177 字节

① MD5算法速度：

```
hash:  
a83af2d2d73e9046ab438c7ba399caac  
耗时: (s)  
9.581597  
速度:  
1.24217119 M/s
```

② RSA算法速度：

```
耗时: (s)  
0.004987001419067383  
速度:  
50.13032461634077Kb/s
```

③ IDEA算法速度：

```
IDEA速度:  
243.83016246618814 KB/s
```

④ Zip算法速度:

```
zip压缩速度  
107.5923457465 KB/s
```

⑤ base64算法速度:

```
base64速度  
197.175623457 KB/s
```

经过多次测速得到平均值，如下所示的各算法的平均速度:

- MD5: 1.1748 MB/s
- RSA: 54.7892 Kb/s
- IDEA: 255.4845 KB/s
- zip: 106.5983 KB/s
- base64: 196.9723 KB/s