ECDSA伪造

一.基本概念

1.ECDSA流程

ECDSA是使用椭圆曲线对DSA数字签名的模拟。流程如下

选择一条椭圆曲线，阶为n，基点为G； 私钥：d (0<d<n)； 公钥：P=dG； 消息：m。

$$
\begin{gathered}
\text{签名过程：} \\
1.\text{生成一个临时密钥} k(0<k<n) \\
2.\text{计算} R=kG=(x,y) \\
3.r=x \\
4.s=k^{-1}(hash(m)+dr) \\
5.(r,s)\text{为签名} \\
\text{验证过程：} \\
1.e=hash(m) \\
2.w=s^{-1}modn \\
3.(x,y)=ewG+rwP \\
4.if(x==r)\text{验证成功} \\
\text{证明：} \\
ewG+rwP=w(eG+rP)=s^{-1}(eG+rP) \\
=k(e+dr)^{-1}(eG+rdG)=kG=R=(x,y)
\end{gathered}
$$

2.ECDSA伪造原理

当ECDSA验证过程中，不检查消息m，而只需要传入m的哈希值e即可验证时，就会引入一种签名伪造方法。具体如下

$$
\begin{gathered}
\text{选择两个整数} u,v \\
\text{计算} R=uG+vP=(x,y) \\
\text{令} r=x \\
\text{令} e=ruv^{-1}modn \\
\text{令} s=rv^{-1}modn \\
\text{则} e \text{为消息哈希值，} (r,s)\text{为伪造的签名}
\end{gathered}
$$

证明也很简单

$$
\begin{gathered}
\text{证明：} \\
s^{-1}(eG+rP)=r^{-1}v(ruv^{-1}G+rP) \\
=uG+vP=R \\
\text{其中} r=R_x, \text{故可以通过验证}
\end{gathered}
$$

3.比特币中的ECDSA

在比特币中，ECDSA使用的椭圆曲线为secp256k1。根据维基百科，secp256k1椭圆曲线的一些参数如下图

与 Koblitz 曲线 secp256k1 关联的 F_p 上的椭圆曲线域参数由六进制 T = （p，a，b，G，n，h） 指定，其中有限域 F_p 定义为：

- p = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFFFF FFFFFFF FFFFFFFFFF FFFFFFFF
- = $2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$

F_p 上的曲线 E：$y^2 = x^3+ax+b$ 定义为：

- a = 000000000 00000000 0000000 00000000 00000000 000000000 000000000 000000000
- b = 000000000 0000000 0000000 000000000 00000000 000000000 00000000 00000007

压缩形式的基点 G 为：

- G = 02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798

和未压缩的形式是：

- G = 04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8

最后，G的n阶和辅因子是：

- n = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141
- h = 01

这是secp256k1的椭圆曲线$y^2 = x^3 + 7$在实数
的图形。请注意，由于 secp256k1 实际上是在
Z_p 上定义的，因此其图形实际上看起来像随机
而不是类似的东西。

其次，在比特币中，所有的哈希函数都使用SHA-256，因此ECDSA中的哈希函数同样是SHA-256，而不是其他ECDSA算法中所使用的SHA-1。

二.代码实现

1.椭圆曲线上的运算实现

要想实现ECDSA，首先要实现一些底层的椭圆曲线上的运算。如下图。

```
class EllipticCurve:
    def __init__(self,p=None,a=None,b=None,g=None,n=None,h=None):...

    def on_curve(self,point):...

    def negative(self,point):...

    def add(self,p, q):...

    def multi(self,k, point):...

    def inverse(self,a, n):...

    def compute_y(self,x):...

    # 求二次剩余
    def quadratic_residue(self,n, p):...
```

其中init函数负责给椭圆曲线类传入一些基本的参数。

on_curve函数判断point点是否在椭圆曲线上，通过是否满足椭圆曲线的方程即可判断。

negative函数可以求-point，若point =(x,y), 则-point=(x,-y)。

add函数可以求p+q，其公式如下图

- 如果$P_3 = (x_3, y_3) = P_1 + P2 \neq O$，

$$\begin{cases} x_3 = \lambda^2 - x_1 - x_2, \\ y_3 = \lambda(x_1 - x_2) - y_1, \end{cases}$$

其中

$$\begin{cases} \lambda = \frac{y_2 - y_1}{x_2 - x_1}, \text{如果} x_1 \neq x_2 \\ \lambda = \frac{3x_1^2 + a_4}{2y_1}, \text{如果} x_1 = x_2 \end{cases}$$

multi函数可以求k*point。通过调用k次加法来实现。

inverse函数可以求a^{-1}modn。通过扩展欧几里得算法实现。

compute_y函数输入x，可以计算出椭圆曲线上的点(x,y)和(x,-y)。通过求解椭圆曲线方程实现，其中需要调用求二次剩余的函数。

quadratic_residue可以求二次剩余，通过Cipolla's algorithm实现。注意p只能为奇素数。

最后，实例化Secp256k1椭圆曲线

```
Secp256k1 = EllipticCurve(
    p=0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f,
    a=0,
    b=7,
    g=(0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798,
0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8),
    n=0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141,
    h=1,
)
```

2.ECDSA类

定义一个ECDSA类，其中生成签名过程与“基本概念”中所描述的一致。但是验证签名不需要检查消息m，只检查消息的哈希值e。

在初始化阶段，只需要传入一个椭圆曲线即可。私钥通过调用ecdsa库中的私钥生成函数自动生成。公钥由私钥计算而来。

```
class ECDSA:
    def __init__(self,curve):
        self.curve = curve
        #自动生成私钥
        sk = ecdsa.SigningKey.generate(curve=ecdsa.curves.SECP256k1)
        self.private_key = int(sk.to_string().hex(),base=16)
        #生成公钥
        self.public_key = self.curve.multi(self.private_key,self.curve.g)
```

在签名生成阶段，需要传入消息message。而后按照“基本概念”中的步骤，通过调用椭圆曲线类中定义的基本运算，即可完成签名的生成。

```python
#签名
    def sig(self,message):
        e = int(SHA256.new(message.encode()).hexdigest(),base=16)
        k = random.randint(0,self.curve.n)
        R = self.curve.multi(k,self.curve.g)
        r = R[0]
        assert r!=0
        k_1 = self.curve.inverse(k,self.curve.n)
        s = (k_1*(e+self.private_key*r)) % self.curve.n
        return r,s
```

在验证签名阶段，需要传入消息哈希message_hash，签名r，s。为了方便后序进行伪造，验证函数可以不使用类中的公钥进行验证，而是可以指定公钥。将choose_public_key设置为True,令public_key等于指定公钥即可。

```python
 #验证签名
    def verf(self,message_hash,r,s,choose_public_key=False,public_key=None):
        if choose_public_key ==True:
            self.public_key = public_key
        e = message_hash
        w = self.curve.inverse(s,self.curve.n)
        (x,y) = 
self.curve.add(self.curve.multi(e*w,self.curve.g),self.curve.multi(r*w,self.publ
ic_key))
        return x==r
```

3.伪造函数

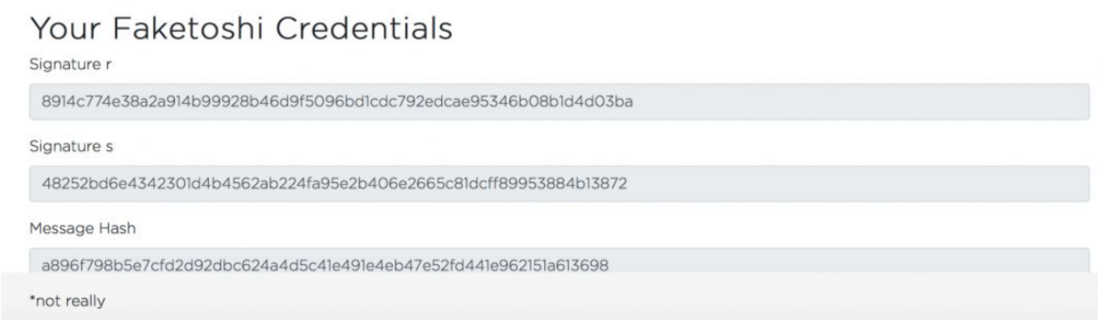
在“基本概念”中，我们可以看到，签名的伪造过程所涉及的参数只有公钥。可以说，对于Alice，只要拿到了Alice的签名公钥，即可调用该算法对Alice的签名进行伪造。因此，伪造签名函数只需要传入一个公钥。同时，注意到，伪造签名算法中u和v的取值可以随意取，而不影响算法的成功性。因此为了方便，在本次实验中，指定u=5，v=7。

同样的，按照“基本概念中的步骤”，调用椭圆曲线类中的运算，即可完成伪造。

```python
def sig_forge(public_key):
    '''伪造签名过程'''
    u = 5
    v = 7
    R = 
Secp256k1.add(Secp256k1.multi(u,Secp256k1.g),Secp256k1.multi(v,public_key))
    r = R[0]
    e = (r*u*Secp256k1.inverse(v,Secp256k1.n))% Secp256k1.n
    s = (r*Secp256k1.inverse(v,Secp256k1.n)) % Secp256k1.n
    return e,r,s
```

4.计算中本聪签名公钥

通过上述代码，我们只需要拿到中本聪的公钥，即可伪造中本聪的签名。但是，我并没有在网上找到创世区块的签名和公钥。所以我只能使用可以通过创世区块验证的签名来倒推中本聪的公钥。可以通过验证的签名如下图



通过签名推导出公钥的过程如下（可以算出两个可能的公钥，真正的公钥为二者之一）

$$\text{已知}r(\text{即}R\text{的}x\text{坐标})\text{，通过椭圆曲线方程可以计算出}R'\text{和}-R'\text{。}$$
$$\text{但无法确定}R'\text{和}-R'\text{哪个是真正的}R\text{，因此令}R'=R1,-R'=R2\text{。}$$
$$\text{同时},s^{-1}(eG+rP)=R,\text{故}P=r^{-1}(Rs-eG)\text{。}$$
$$\text{通过}R1\text{，}R2\text{，可以计算出两个公钥，记为}pk1\text{，}pk2\text{。}$$

具体代码如下。先计算出R1，R2，若两者都在椭圆曲线上，算法继续。而后按照上述方法计算pk1和pk2，并返回。

```
def compute_public_key(curve:EllipticCurve,message_hash,r,s):
    '''基于签名恢复出公钥'''
    G = curve.g
    e = message_hash
    R1,R2 = curve.compute_y(r)
    assert curve.on_curve(R1)
    assert curve.on_curve(R2)
    r_1 = curve.inverse(r,curve.n)
    eG = curve.multi(-e,G)
    Rs1 = curve.multi(s,R1)
    Rs2 = curve.multi(s,R2)
    public_key1 = curve.multi(r_1,curve.add(Rs1,eG))
    public_key2 = curve.multi(r_1,curve.add(Rs2,eG))
    return public_key1,public_key2
```

三.结果分析

在主函数中进行测试，代码如下。分别对计算出的公钥1和公钥2执行伪造签名的算法，并输出。值得注意的是，在ECDSA中，公钥是一个点，拥有x，y坐标。但是一般来说，输出公钥格式为0x04+<公钥x坐标>+<公钥y坐标>。因此在测试函数中也采用同样的方法进行输出。

```
if __name__=="__main__":
    test = ECDSA(Secp256k1)

    message_hash = 
"a896f798b5e7cfd2d92dbc624a4d5c41e491e4eb47e52fd441e962151a613698"
```

```
    r = "8914c774e38a2a914b99928b46d9f5096bd1cdc792edcae95346b08b1d4d03ba"
    s = "48252bd6e4342301d4b4562ab224fa95e2b406e2665c81dcf89953884b13872"
    pk1,pk2 =
compute_public_key(Secp256k1,int(message_hash,base=16),int(r,base=16),int(s,base
=16))
    prefix = "0x04"
    print(f"计算出的公钥1为(十进制):{prefix+str(hex(pk1[0]))[2:]+str(hex(pk1[1]))
[2:]}")
    print(f"计算出的公钥2为(十进制):{prefix+str(hex(pk2[0]))[2:]+str(hex(pk2[1]))
[2:]}")
    print("----使用公钥1----")
    e, r, s = sig_forge(pk1)
    print("伪造的签名为：")
    print(f"消息哈希：{hex(e)}")
    print(f"签名r:{hex(r)}")
    print(f"签名s:{hex(s)}")
    print(f"验证结果为{test.verf(e, r, s, choose_public_key=True,
public_key=pk1)}")

    print("----使用公钥2----")
    e, r, s = sig_forge(pk2)
    print("伪造的签名为：")
    print(f"消息哈希：{hex(e)}")
    print(f"签名r:{hex(r)}")
    print(f"签名s:{hex(s)}")
    print(f"验证结果为{test.verf(e, r, s, choose_public_key=True,
public_key=pk2)}")
```

结果如下。可以看到，伪造的两组签名都可以通过签证，实验成功。

```
ECDSA_Class
"C:\merkle tree\venv\Scripts\python.exe" "C:/merkle tree/ECDSA_Class.py"
计算出的公钥1为(十进制):0x04d9386358a4b382dcc308e99577c604c553003e304e3b835ec4e55c47cd1beddb3a54de1b820843facfb244d7d6b68a353643da61729d06c982836cf14b95
计算出的公钥2为(十进制):0x0474c8e223a318b586bd13bfa3c5224ba9883686bff1c680d586bed86c73b902dc47559e52e4b184f0f09055d6d7b2464d58f98966d73c3ad53f574ff9ccc6
----使用公钥1----
伪造的签名为:
消息哈希：0xb12c86dfdf9e3d9ca01b24b58436262e80ed8c219d86eaeb06412438fcdd7532
签名r:0xc4d7f0063910bca81359336485e568a7c25ccb341fe18f3d159786339ec4ca39
签名s:0x89d5b4932cb93f85b99f075780d7a13bfe0f0dfc98d1a246e7946043b90ecb24
验证结果为True
----使用公钥2----
伪造的签名为:
消息哈希：0x9ab8e5b1582543be865992c9312f6fc9a47bfe602dc0057ef85926cebaa0b23c
签名r:0xbcfa7f848342ba455b09a4cde759c81b754e69ae705ede85bd4844a5eb59220
签名s:0xb88b6123780773f2e7ab83c1d6a3165ac41b51040beb947071900d4aa240b133
验证结果为True

Process finished with exit code 0
```

参考博客:

[1]椭圆曲线上点的运算 - 灰信网（软件开发博客聚合）(freesion.com)

[2](48条消息) 二次同余方程（二次剩余）*胡牧之.的博客-CSDN博客*二次同余方程

[3]Faketoshi签名工具在手，人人都是「中本聪」 - 碳链价值 (ccvalue.cn)