课程作业

Crypto CSY

1 Merkle Tree

1.1 Implement Merkle Tree as of RFC 6962 in any programming language that you prefer

编程语言：Python

实现思路：采用多维数组（多维列表）存储并实现Merkle树。Merkle树从下至上的每一层节点的哈希值依次存储至多维列表中，详情代码见附录。

生成Merkle树过程：

(1)该层节点为偶数个节点时，会两两依次配对，加前缀级联生成父节点的哈希值

(2) 当该层节点为奇数个节点时，最后一个节点会直接作为父节点，其余节两两依次配对，加前缀级联生成父节点的哈希值。

(3) 每一层节点都由靠近叶节点的底下一层生成。按照上述规律进行节点生成，直到生成根节点（该层有且仅有一个节点）

(4) 注意按照RFC 6962的标准要求，叶节点和其余节点的进行哈希的前缀不同（叶节点前缀为0x00，其余节点前缀为0x01）

```
... 
========== RESTART: C:\Users\Dragon\Desktop\Merkle\merkle_tree1.0.py
节点: a, b, c, d, e, f, g
Merkle Tree的高度: 3
Merkle Tree从底至上每层节点哈希值:
第1层:
022a6979e6dab7aa5ae4c3e5e45f7e977112a7e63593820dbec1ec738a24f93c,
57eb35615d47f34ec714cacdf5fd74608a5e8e102724e80b24b287c0c27b6a31,
597fcb31282d34654c200d3418fca5705c648ebf326ec73d8ddef11841f876d8,
d070dc5b8da9aea7dc0f5ad4c29d89965200059c9a0ceca3abd5da2492dcb71d,
2824a7ccda2caa720c85c9fba1e8b5b735eecfdb03878e4f8dfe6c3625030bc4,
f5a06d3c52937089c51b7c6c1cc1948ccdc5581328b2ebb578e8cca66a7b5221,
5aeb196e83598231b45c61f3e0c5a0fda49b0d4f86a6db5f893aacccf514fa99

第2层:
4c64254e6636add7f281ff49278beceb26378bd0021d1809974994e6e233ec35,
40e2511a6323177e537acb2e90886e0da1f84656fd6334b89f60d742a3967f09,
a3e85c50b3d0f7f2dfcbed0927df54902f37a2bc46ec305195989ef2b290a4bf,
5aeb196e83598231b45c61f3e0c5a0fda49b0d4f86a6db5f893aacccf514fa99

第3层:
9dc1674ae1ee61c90ba50b6261e8f9a47f7ea07d92612158edfe3c2a37c6d74c,
f6c3f94f722c14e192e11573410dd81e9837726af844f8b0975ad4ff050768bf

第4层:
e9cb499ef643bc53fcfa08828d00af59c53c3fd57761a95669d68771f2ff6cef
```

图 1: Implement Merkle Tree运行截图

1.2 Construct a merkle tree with 100k leaf nodes

按照上一问的生成方法，构造大小为100k的Merkle树。Merkle树的深度为17，运行结果如下：

```
#100k test
lst = []
for i in range(100000):
    lst.append(str(i))
merkle_tree,h = Create_Merkle_Tree(lst)
```

```
. . .
========== RESTART: C:\Users\Dragon\Desktop\Merkle\merkle_tree1.0.py
>>> Show_Merkle_Tree(merkle_tree,h)
Merkle Tree的高度: 17
Merkle Tree从底至上每层节点哈希值:
Squeezed text (100001 lines).
Squeezed text (50001 lines).
Squeezed text (25001 lines).
Squeezed text (12501 lines).
Squeezed text (6251 lines).
Squeezed text (3126 lines).
Squeezed text (1564 lines).
Squeezed text (783 lines).
Squeezed text (392 lines).
Squeezed text (197 lines).
Squeezed text (99 lines).
Squeezed text (50 lines).
第13层:
94e619642bd3b8f226af5a6c875a1c5146c2f4a9529ec8b6e987fc40d84771dc,
94fc7230afa396376019869b730af24211628d528e19ec0cc30a598c485f8060,
94f23c1748d56c5445820b2f663caef486ee32de24f376eb884e3957f6d806c4,
2a3c736059190514b1d8997b4a3c67f0bf50728ca9bcf28a63db0ff715caf53f,
13d0c25a9d7757f799ddd659c6b284cdbf9a502b1f0cbe2325b5e4990cc8d902,
f0b26ab077ab211e666af739488f18b52bf8b0909f0152cfa03cca23155ab23a,
5f2a7d6738768089a5581f6cbe4e85768446fb66306e3a897ebaaaf571ffe5e0,
bdc9fb1ed32cd1565bd72ff380cdfd7a72d8fbf8cd4c5f3e373c6fe3587bcebf,
c9ef6cf1df819c800ceff3b3a2bd89aaab20ddc4bd78670d5d2f5af938d8f5ef,
```

图 2: 100k叶节点测试运行截图

1.3 Construct the existence (inclusion) proof for randomly chosen leaf node & verify the proof

对特定的Merkle树，指定需要查找的节点和节点序号，判断该节点是否在Merkle树中。

(1)首先根据提供的节点，在底层计算叶子节点的哈希值；根据节点序号n的奇偶性判断查找的节点为$n-1$还是$n+1$，注意如果该层节点个数为奇数且节点为最后一个，父节点为其本身；最后将n的值取半，n的新值恰好为生成的父节点，可据此进行下一层的查找计算操作.

(2)重复操作，依次向上查找。最终判断生成的哈希值是否与根节点相同：若相同则在Merkle树中，反之不在。查找过程的关键代码如下：

```
while j<h:
    L = len(merkle_tree[j])
    if L%2 == 1 and L-1==n:
        pass
    elif n%2==1:
        hash_value = hash_node(merkle_tree[j][n-1]+hash_value,hash_function)
    elif n%2==0:
        hash_value = hash_node(hash_value+merkle_tree[j][n+1],hash_function)
    n = n//2
    j += 1
```

```
========== RESTART: C:\Users\Dragon\Desk
要查找的节点：23
节点序号：23
序号:23，字符:23
查找路径:
第1层查询值:15389e67ab423d24db388cf88c92a3e07c540e3342b7b2fa3ea75e942c8bd213，
        生成的Hash值:36f9904b0a27c51ed995025f0cce64883d72d0844b2ca566fffba481c010908a
第2层查询值:3dccaa425c6ab67bd8527a1cf26a3ea8180d3e3707496921d3edb7680e38dcbc，
        生成的Hash值:441abababeb97f088be1da9239bf1c32e4a220098b0c5ea7d61a93c10432a9aa5e
第3层查询值:0dfeb2314186e2961145704e4defe22d9d8a52be10056ba028c05239c91f4f4f，
        生成的Hash值:980d7ae3084bc62e4f6928314e7be79a04c3532897ba5b774c7f7fcd72e4b684
第4层查询值:a230f50723789826f04ee4048d32814b90a4bdfbeecdf7759478ee1794acf64f，
        生成的Hash值:27b5ea5265b18f9938dac41ab31df24c91ab778013205c3fb50d07a365d84763
第5层查询值:19040f40a98aa82b160bd63a2d1a3ccf1421ee19d9b339b90cff8f66e8830525，
        生成的Hash值:5c065ee189c121f6fbb998ad1a8f55c70329b0bf6dfdba84489599dc9408366c
第6层查询值:20bc8c70d66d426524f269001cc47c70f0a53f39f09f016127f75d5176987da2，
        生成的Hash值:08fe4160949301a5fb321bd2895c58c840e098d46f6722f449c1c8025ccee120
第7层查询值:64cae0469a19ec5b7f30b6f016b912970c564df616e5b0d4fe1c7fe9a945df2f，
        生成的Hash值:8ebc573ac215a2d37ada2e57674bada9f2ce878d2afc24058adc45fc3941a9e9
第8层查询值:ad1e67a7450e51b6997eed94b20bbdb50aa8a2348bcb55988b1936671bfffda7，
        生成的Hash值:0221f1ad57c0419bb2a105d7bd665b436cd0657e82d3d56ca1d00d5f57c967c5
第9层查询值:51e255e8c1020898e3bd38263ecd905d3955e828c964a7f7eaf74074031cada6，
        生成的Hash值:b7fa906b79abbc56babf2d520a52e58d8973833c49cfa2ff9491347e4cb3510d
第10层查询值:b11f458b4958ddcc2b50b02982ffe79176550f496071b8b37ee5e767336a3109，
        生成的Hash值:09c375d93c0d343385ef14a9e0fe5c567ed52b6d24a027618409ef1134dad544
第11层查询值:1fa7bbddab3b0f1ca126a18efa4bb02ff0290eb9b2e003648d1a222d015ba5ca，
        生成的Hash值:6db5f9c75bd50236213ad641ceac77f2d56d496fb0f70d00d48d70a31d5066cc
第12层查询值:a0c101c792e52e6214150849c2f192380e14d86a4c1db147f4ce43d146c3bb16，
        生成的Hash值:94e619642bd3b8f226af5a6c875a1c5146c2f4a9529ec8b6e987fc40d84771dc
第13层查询值:94fc7230afa396376019869b730af24211628d528e19ec0cc30a598c485f8060，
        生成的Hash值:c089da8360be0503314367c6dfabb2a2be25d327456703eb310208f2fb72e8e9
第14层查询值:c5be2acd278fd317eadc1e984a41415f18893bf21d1fd96afcf7959308543ff8，
        生成的Hash值:86bbefecf4d9b38a73ea799976338840b0afa35abb51026de1725524bd64bda3
第15层查询值:e187d4d73a3c14bd9e36f32f172ed8e0c916243cc50b3aabc035b9515541bc39，
        生成的Hash值:caf1c034fdd6e533cfc6e9c484249984lec99aea9127f894ead533dc9c4f3b70
第16层查询值:1504312e5d90ffea4c8de94ad5e04465ffb509c29e254e498433ee20dc74e51c，
        生成的Hash值:d5db17981bbae0f753fc66ac04ecb1badb6b09a954f38ad5ea584a9549a34060
第17层查询值:9940b80b01132e06e7111c92a6095d52afda304cf05f2578016311a8b8710ba5，
        生成的Hash值:6191ccc9f8fbe88c36b105e600a3bb45e955a5a7468d00af9b0cbdc98705d9f5

根节点哈希值: 6191ccc9f8fbe88c36b105e600a3bb45e955a5a7468d00af9b0cbdc98705d9f5
节点23在Merkle树中
```

图 3: 等于根节点情况

```
========== RESTART: C:\Users\Dragon\Desk
要查找的节点：23
节点序号：30
序号:30，字符:23
查找路径:
第1层查询值:5c9e5b4f30fef838895b0d4c0b301771562edcf838bd9af3199480f467e2bcf0，
        生成的Hash值:eabdeac25d9705a31a4387b29e3f98ee37c7f47c17b5ab0afd9ae1932512a828
第2层查询值:967ea7b2a2d3f970c042b31402cc919126912b97db819db862dfad6d233c637f，
        生成的Hash值:f31ceb8d8233cf1e84eeda7898aae34ceabfee7ca119b752bd74d37aa16863d4
第3层查询值:83962f83c00e13b56ed612ae1c52eab0f4d001357303f01b6afb0e6a5b6c5102，
        生成的Hash值:e58bf02f3d964e98f8d7035a66d5c73fd4d21e637e612b8c348fa28e6c93cbbe
第4层查询值:980d7ae3084bc62e4f6928314e7be79a04c3532897ba5b774c7f7fcd72e4b684，
        生成的Hash值:8163f2c7036f99e2e68e13e0ad166fd45bac81c21c7696d2869464f0dad0ec26
第5层查询值:19040f40a98aa82b160bd63a2d1a3ccf1421ee19d9b339b90cff8f66e8830525，
        生成的Hash值:c7d112d2930718968c90a71dfba47638ee062ba4a2605810f4a27a003cff7683
第6层查询值:20bc8c70d66d426524f269001cc47c70f0a53f39f09f016127f75d5176987da2，
        生成的Hash值:386a242f47e459bc6288358e5af8efd6b4e15131299862ced50138cadfbe56e8
第7层查询值:64cae0469a19ec5b7f30b6f016b912970c564df616e5b0d4fe1c7fe9a945df2f，
        生成的Hash值:2b10922433a7f37a03cf342d9492f9982c3b0fd963761c2835d5551ba8c17942
第8层查询值:ad1e67a7450e51b6997eed94b20bbdb50aa8a2348bcb55988b1936671bfffda7，
        生成的Hash值:a025ff049572142f7185dd029b8f03b2e3d04853bfa1c74199415851005de463
第9层查询值:51e255e8c1020898e3bd38263ecd905d3955e828c964a7f7eaf74074031cada6，
        生成的Hash值:a746c2332999a0e89454d20ceb7b292b184994d7d6ec1326ce955f1cf1c8a26a
第10层查询值:b11f458b4958ddcc2b50b02982ffe79176550f496071b8b37ee5e767336a3109，
        生成的Hash值:87bca02d8e86c4a70032c8908d8b312749a6fb9b47836921b2c26d98d8a5ebce
第11层查询值:1fa7bbddab3b0f1ca126a18efa4bb02ff0290eb9b2e003648d1a222d015ba5ca，
        生成的Hash值:6564bf6e0029c9ddc74ac75e41cfed7fa076388d7e40b3c71762d831cc5ffc3f
第12层查询值:a0c101c792e52e6214150849c2f192380e14d86a4c1db147f4ce43d146c3bb16，
        生成的Hash值:3974f350b3c76c3ccd6a2cabcb5f7f8a5976fe582d84ae58d4ccf89720c580e1
第13层查询值:94fc7230afa396376019869b730af24211628d528e19ec0cc30a598c485f8060，
        生成的Hash值:6c999cb6b88779665e2267818617deaad344fc3d526790e70f50bd3aaebc3b53
第14层查询值:c5be2acd278fd317eadc1e984a41415f18893bf21d1fd96afcf7959308543ff8，
        生成的Hash值:fcf911b496dc9d752f95da9f2fcac5656e1664d6e9e072c5be366a180eb7f41d
第15层查询值:e187d4d73a3c14bd9e36f32f172ed8e0c916243cc50b3aabc035b9515541bc39，
        生成的Hash值:cec1e4bad08aa94e2034721e137d23a67904547158f1c15fd50d4387782cc311
第16层查询值:1504312e5d90ffea4c8de94ad5e04465ffb509c29e254e498433ee20dc74e51c，
        生成的Hash值:1bf44ce7ee0f5ac23f34d96e5bd77cdf24661cf4f36de7c8be78d9b364d6126b
第17层查询值:9940b80b01132e06e7111c92a6095d52afda304cf05f2578016311a8b8710ba5，
        生成的Hash值:5555d3bfdba1431df35e1d2f5322eb7c458f9c0100ea358c7fa0450e11c3ab55

根节点哈希值: 6191ccc9f8fbe88c36b105e600a3bb45e955a5a7468d00af9b0cbdc98705d9f5
节点23不在Merkle树中
```

图 4: 不在Merkle树情况

需要判定查找的节点序号是否小于叶节点个数，如果大于叶节点个数则报错：

```
==================
要查找的节点：a
节点序号：100001
节点序号有误!
```

图 5: 错误提示

2 ECDSA

椭圆曲线算法是基于有限域上椭圆曲线所形成的循环子群上。因此，算法需要以下几个重要参数[2]

- 素数p，用于确定有限域的范围
- 椭圆曲线方程参数a和b
- 用于生成子群的基点G
- 子群的阶n
- 辅助因子h In conclusion, the domain parameters for our algorithms are the sextuple (p, a, b, G, n, h).

具体实现中，采用标准化的椭圆曲线secp256k1，主要参数如下[3]：

$$
\begin{aligned}
p &= FFFFFFFF\ FFFFFFFF\ FFFFFFFF\ FFFFFFFF \\
&\quad FFFFFFFF\ FFFFFFFF\ FFFFFFFE\ FFFFFC2F \\
&= 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1 \\
E &= y^2 = x^3 + ax + b \text{ over } \mathbb{F}_p \\
a &= 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000 \\
b &= 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000007 \\
G &= 04\ 79BE667E\ F9DCBBAC\ 55A06295\ CE87OB07\ 029BFCDB\ 2DCE28D9\ 59F2815B\ 16F81798 \\
&\quad 483ADA77\ 26A3C465\ 5DA4FBFC\ OE1108A8\ FD17B448\ A6855419\ 9C47D08F\ FB1OD4B8 \\
n &= FFFFFFFF\ FFFFFFFF\ FFFFFFFF\ FFFFFFFE \\
&\quad BAAEDCE6\ AF48A03B\ BFD25E8C\ D0364141 \\
h &= 01
\end{aligned}
$$

自定义椭圆曲线六元组；根据标准文档设置参数值；定义椭圆曲线上的运算和签名验签操作：

模逆 采用扩展欧几里得方法求模逆$x^{-1} \mod p$. 注意x不为0；且当$x < 0$时，要将其化成在有限域$\mathbb{F}_n$上的形式进行运算.

是否在椭圆曲线上 根据椭圆曲线方程判断点是否在椭圆曲线$E(F_p)$上.

点加 椭圆曲线 $E(F_p)$ 上的点按照下面的加法运算规则,构成一个交换群:

a) $O + O = O$;

b) $\forall P = (x, y) \in E(F_p) \backslash \{O\}, P + O = O + P = P$;

c) $\forall P = (x, y) \in E(F_p) \backslash \{O\}, P$ 的逆元素 $-P = (x, -y), P + (-P) = O$;

d）两个非互逆的不同点相加的规则：设 $P_1 = (x_1, y_1) \in E(F_p) \backslash \{O\}, P_2 = (x_2, y_2) \in E(F_p) \backslash \{O\}$, 且 $x_1 \neq x_2$, 设 $P_3 = (x_3, y_3) = P_1 + P_2$, 则

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

其中

$$
\lambda = \frac{y_2 - y_1}{x_2 - x_1}
$$

e）倍点规则: 设 $P_1 = (x_1, y_1) \in E(F_p,) \backslash \{O\}$, 且 $y_1 \neq 0, P_3 = (x_3, y_3) = P_1 + P_1$, 则

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

多倍点 椭圆曲线上同一个点的多次加为该点的多倍点运算。设k是一个正整数，P是椭圆曲线上的点，称点P的k次加为点P的k倍点运算，记为$Q = [k]P = \underbrace{P + P + \cdots + P}_{k个}$。

生成公私钥对 私钥d在$\{1, \cdots, n-1\}$范围内任取一值，公钥$P = dG$

签名 $k \leftarrow Z_n^*, R = kG$

$r = R_x \bmod n, r \neq 0$

$e = \text{hash}(m)$

$s = k^{-1}(e + dr) \bmod n$

Signature is (r, s)

验签 $e = \text{hash}(m)$

$w = s^{-1} \bmod n$

$(r', s') = e \cdot wG + r \cdot wP$

Check if $r' == r$

```
========== RESTART: C:\Users\Dragon\Desktop\ECDSA\ecdsa1.0.py ==========
Curve: secp256k1
Private key: 0xf8919647497eaa6a80193acb3b21d522f424d1e9d8746e0da0d8fd0f4eed4c4f
Public key: (0x87867a98ed07e1ae35db9ac2995f712fa7888849ddbf2be8409c88084f18909, 0xa2995f32f0660fc7a567b8793b2a91e937b9aa6fb01d3637f76fc41ef265a778)

Message: b'How u doing?'
Signature: (0xc1b15d0c7994036448ea888db3da0379a17349335f31bd1ab75df67bc73d4322, 0x7d58fce35279ade121af9581d6b7fda3548109c6db0338bb41e34883191b40f1)
Verification: signature matches
```

图 6: ECDSA签名验签

2.1 Leaking k leads to leaking of d

通过设置全局变量global k_leak模拟k泄露的情况，可以通过k和一次签名恢复密钥$d = r^{-1}(ks - e) \bmod n$,具体推导过程和关键代码如下：

$$s = k^{-1}(e + dr) \bmod n$$
$$ks = e + dr \bmod n$$
$$d = r^{-1}(ks - e) \bmod n$$

```
1    d_guess = (inverse_mod(r, curve.n)*(k_leak*s - e))%curve.n
2    print('猜测密钥d = {0}'.format(hex(d_guess)))
3    if d_guess == d:
4        print('Success!')
5    else:
6        print('Failed.')
```

```
(1) Leaking k leads to leaking of d:
泄露的k = 0x8e5225bd7190d6b20b8608de111dd4e0a34a4b485d6861f0df13d5d68802de12
猜测密钥d = 0xf8919647497eaa6a80193acb3b21d522f424d1e9d8746e0da0d8fd0f4eed4c4f
Success!
```

图 7: Leaking k leads to leaking of d

2.2 Reusing k leads to leaking of d

两次签名运用相同的k，有$r_1 = r_2$。可以通过两次签名(r_1, s)和(r_1, s_2)求出私钥，推导过程如下：

$$s_1 = k^{-1}(e_1 + r_1 d) \bmod n, \quad s_2 = k^{-1}(e_2 + r_1 d) \bmod n$$

$$\frac{s_1}{s_2} = \frac{e_1 + r_1 d}{e_2 + r_1 d} \bmod n \rightarrow d = \frac{s_1 e_2 - s_2 e_1}{s_2 r_1 - s_1 r_1} \bmod n$$

关键代码和运行结果：

```
d_guess2 = ((s2*e1-s1*e2)*inverse_mod(s1*r1-s2*r1, curve.n)) %curve.n
print('猜测密钥d = {0}'.format(hex(d_guess2)))
if d_guess2 == d:
    print('Success!')
else:
    print('Failed.')
```

```
(2) Reusing k leads to leaking of d:
重复使用的k = 0x8e5225bd7190d6b20b8608de111dd4e0a34a4b485d6861f0df13d5d68802de12
猜测密钥d = 0xf8919647497eaa6a80193acb3b21d522f424d1e9d8746e0da0d8fd0f4eed4c4f
Success!
```

图 8: Reusing k leads to leaking of d

2.3 Two users, using k leads to leaking of d, that is they can deduce each other' s d

Alice用私钥d_1签名消息m_1：$\sigma_1 = (r, s_1)$ $s_1 = k^{-1}(e_1 + rd_1) \bmod n$ $e_1 = \text{hash}(m_1)$

Bob用私钥d_2签名消息m_2：$\sigma_2 = (r, s_2)$ $-s_2 = k^{-1}(e_2 + rd_2) \bmod n$ $e_2 = \text{hash}(m_2)$

因此：

$$k = s_1^{-1}(e_1 + rd_1) \bmod n \quad k = s_2^{-1}(e_2 + rd_2) \bmod n$$

$$s_1(e_2 + rd_2) = s_2(e_1 + rd_1) \bmod n$$

Alice可以计算Bob的密钥 $d_2 = (s_2 e_1 - s_1 e_2 + s_2 r d_1) / (s_1 r) \bmod n$

Bob同样可以计算Alice的密钥 $d_1 = (s_1 e_2 - s_2 e_1 + s_1 r d_2) / (s_2 r) \bmod n$

```
#r1=r2
#Alice 恢复的密钥Bob
d_guess_Bob = ((s_2*e_1-s_1*e_2+s_2*r_1*d1)*inverse_mod(s_1*r_1,curve.n)) % curve.n
print('恢复的私钥AliceBob:',hex(d_guess_Bob))
if d_guess_Bob == d2:print('Success!')
else:print('Failed.')

#Bob 恢复Alice 的密钥
d_guess_Alice = ((s_1*e_2-s_2*e_1+s_1*r_1*d2)*inverse_mod(s_2*r_1,curve.n)) % curve.n
print('恢复的私钥BobAlice:',d_guess_Alice)
if d_guess_Alice == d1:print('Success!')
else:print('Failed.')
```

```
(3) Leaking Secret Key Via Reusing k By Different User:
Alice的私钥d1: 0x793abf9a328a969b7d5e0d9debe5715db3ed4c98f882ef7f95ad416f685be0ae
Alice的公钥P1: (0xea22ac0ece66be3ebd3825bb2e80f495c19833c8fc59a33892f069303d01dcd3, 0x614a0dd4f154540c046b56f02cc49c01eab3e4a292430cca741b3bd25095f8cd)
Alice签名的消息: b'ShanDong'

Bob的私钥d2: 0xa2bbb141c013e73528996612fe44ca4926989b09971c15d6a84a615560208bd1
Bob的公钥P2: (0x94263905881185c3b09c4f83d31c07390324c571428a11983ab3a49e2a6a6d21, 0xece2a8adcf4bc2826e368c596a9f1783c97deec92257aa8697b776fbfdebe42a)
Bob签名的消息m2: b'QingDao'

Alice恢复Bob的私钥: 0xa2bbb141c013e73528996612fe44ca4926989b09971c15d6a84a615560208bd1
Success!
Bob恢复Alice的私钥: 0x793abf9a328a969b7d5e0d9debe5715db3ed4c98f882ef7f95ad416f685be0ae
Success!
```

图 9: Two users, using k leads to leaking of d, that is they can deduce each other’s d

2.4-2.6将采用安全机制，每一次随机生成k。方便起见，具体实现直接调用自己模块MyECDSA.py

2.4 Malleability of ECDSA, e.g. (r,s) and (r,-s) are both valid signatures

如果(r,s)通过验证： $e\cdot s^{-1}G+r\cdot s^{-1}P=(x',y'),r=x' \bmod p$;
$(r,-s)$同样通过验证：$e\cdot(-s)^{-1}G+r\cdot(-s)^{-1}P=-\left(e\cdot s^{-1}G+r\cdot s^{-1}P\right)=(x',-y'),r=x' \bmod p$

```
s_neg = -s%curve.n
#(r,s)通过验证，(r,-s)同样可以
signature_forge = (r,s_neg)
print('(r,-s) Verification:', verify_signature(P, msg, signature_forge))
```

```
============== RESTART: C:\Users\Dragon\Desktop\ECDSA\ecdsa2.0.py ==============
(4) (r,s),and (r,-s), are both valid signatures:
消息: b'ShanDong University'
密钥d: 0x6377057c1eddcc9d392d1d634ab5a9678fceaedf47d076dffa9f8fa96f816f6a
公钥P: (0xf1464611cb1fd9221469e3819a67af33fe34639d441623794050e373fdbf0c7a, 0xd5a76fa9bbd00ea7d2c67f9ea306bf1c9997e065453bda351133c364e4edf73e)
Signature: (0xbff80436d4d5f1351b955d70681d5bd02b24a324cbea69d50abf6f51c1cc7d4b, 0xe9980a2c0ff6b413cbb9352e4cae35fcbf88f5e23e15dd03708c814566635a27)
(r,s) Verification: signature matches
(r,-s) Verification: signature matches
```

图 10: Two users, using k leads to leaking of d, that is they can deduce each other’s d

2.5 Pretend to be satoshi as one can forge signature if the verification does not check m

如果协议不对消息m进行验证，攻击者可以伪造消息的哈希以及对应的签名
(1)随机选择 $u,v\in\mathbb{F}_n^*$
(2)计算$R'=(x',y')=uG+vP$
(3)$r'=x' \bmod n,\quad s'=r'v^{-1} \bmod n,$ 伪造消息的哈希值$e'=r'uv^{-1} \bmod n$
(r',s')是私钥d，消息哈希e'的有效签名

```
u = random.randrange(1, curve.n)
v = random.randrange(1, curve.n)
(x,y) = point_add(scalar_mult(u,curve.g),scalar_mult(v,P))
r_ = x%curve.n
e_ = r_*u*inverse_mod(v,curve.n)
s_ = r_*inverse_mod(v,curve.n)

#Check 可以伪造出哈希值为的消息签名e_
w = inverse_mod(s_, curve.n)
u1 = (e_ * w) % curve.n
u2 = (r_ * w) % curve.n
```

```
    (r_forge,s_forge)=point_add(scalar_mult(u1,curve.g),
                                scalar_mult(u2,P))
    if r_forge % curve.n == r_:
        print('Success!')
```

```
(5) one can forge signature if the verification does not check m:
Success!
```

图 11: one can forge signature if the verification does not check m

2.6 Same d and k used in ECDSA & Schnorr signature, leads to leaking of d

Schnorr 签名过程：

(1)采用和ECDSA相同的k, $R = kG$

(2)$e_2 = h(R \parallel m)$

(3)$R = kG, \quad s_2 = (k + e_2 d) \bmod n$

(4)签名(R, s_2)

两种签名方式使用相同的k和密钥d，有：

$$s_1 = (e_1 + r_1 d)\,(s_2 - e_2 d)^{-1} \bmod n$$

$$d = \frac{s_1 s_2 - e_1}{(s_1 e_2 + r_1)} \bmod n$$

```
    #ECDSA
    m = b'ECDSA'
    e1 = hash_message(m)
    R_x, R_y = scalar_mult(k, curve.g)#R=kG
    r1 = R_x % curve.n
    s1 = ((e1 + r1 * d1) * inverse_mod(k, curve.n)) % curve.n

    #Schnoor signature with same private key d
    R = scalar_mult(k, curve.g)
    R_x = hex(R_x)[2:]; R_y = hex(R_y)[2:]
    R = R_x+R_y #注意对 R 进行哈希的方式

    e2 = hash_message(R.encode('utf-8')+m)
    s2 = (k+e2*d)%curve.n

    s1 = ((e1 + r1 * d1) * inverse_mod(s2-e2*d1, curve.n)) % curve.n
    d_guess = (s1*s2-e1)*inverse_mod(s1*e2+r1,curve.n)%curve.n
    if d_guess == d1:
        print('Success!')
```

```
(6) Same d and k used in ECDSA & Schnoor signature, leads to leaking of d:
Success!
```

图 12: same d and k used in ECDSA & Schnorr signature, leads to leaking of d

3 Bonus

3.1 PoC impl of ec point compression

椭圆曲线的对称性：已知x，有且仅有两个y满足在椭圆曲线上。因此，可以用额外的1比特来表示y点。按照标准文档的要求，用前缀+x坐标+y坐标表示椭圆曲线上的点。即不再用64byte，改用33byte(前缀1byte)表示一个点。一共有3种情形表示一个点：

未压缩：04+x坐标+y坐标

压缩，且y是奇数：03+x坐标+y坐标

压缩，且y是偶数：02+x坐标+y坐标

根据点的y坐标奇偶性，加上相对应的前缀。在解压缩时，需要根据x求解y的值并进行判定，即求解一个二次剩余问题。但实际上有更为高效的$Cipolla$算法进行求解。

```python
def EC_Compress(P):#点压缩
    x = P[1]
    if x%2 == 0:
        compressed_point = '02'+hex(P[0])[2:]
    return compressed_point
    elif x%2 == 1:
        compressed_point = '03'+hex(P[0])[2:]
    return compressed_point

#Cipolla Algo
def EC_Uncompress(compressed_point): #点解压缩
    p_hex = 'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F'
    p = int(p_hex, 16)
    x_hex = compressed_point[2:66]
    x = int(x_hex, 16)
    prefix = compressed_point[0:2]

    y_square = (pow(x, 3, p) + 7) % p
    y_square_square_root = pow(y_square, (p+1)//4, p)
    if (prefix == "02" and y_square_square_root & 1) or (prefix == "03" and not
        y_square_square_root & 1):
        y = (-y_square_square_root) % p
    else:
        y = y_square_square_root

    computed_y_hex = format(y, '064x')
    computed_uncompressed_key = "04" + x_hex + computed_y_hex

    return computed_uncompressed_key
```

```
========== RESTART: C:\Users\Dragon\Desktop\ECDSA\Bonus_Compress.py ==========
(1) EC point compression:
点P: 0xdac667999f3d80cd0ebd70984828f67be0c2e53bedfea087849bb852f6b339af 0x7b604f317ee28fa3be8913d56962fed13507d086116558761eff3eb20fb14213
Compress: 03dac667999f3d80cd0ebd70984828f67be0c2e53bedfea087849bb852f6b339af
Uncompress: 04dac667999f3d80cd0ebd70984828f67be0c2e53bedfea087849bb852f6b339af7b604f317ee28fa3be8913d56962fed13507d086116558761eff3eb20fb14213
```

图 13: PoC impl of ec point compression

3.2 PoC impl of extracting public key from signature & verify the signature

通过签名结果来和消息摘要，计算出签名私钥对应的公钥。已知如下关系式：

$$\begin{aligned} R &= kG \\ &= k(e+dr)^{-1}(e+dr)G \\ &= s^{-1}(eG+rP) \end{aligned}$$

所以$P = r^{-1}(sR - eG)$.在这里s, r, G, e均已知，问题转化为求R点，已知r是R点的x坐标。在椭圆曲线上，已知x坐标求y坐标，y有两个不同解，也就可以恢复出两个不同的公钥，其中必有一个为签名者使用的公钥。进行签名认证时需要用两个公钥认证两次。

```
#Extracting public key from signature
#关键在于求 R
y = Secp256k1GetYByX(r)

R1 = (r,y[0])
P_guess1 = scalar_mult(inverse_mod(r,curve.n),
          point_add(scalar_mult(s, R1),
          point_neg(scalar_mult(e,curve.g))));

R2 = (r,y[1])
P_guess2 = scalar_mult(inverse_mod(r,curve.n),
          point_add(scalar_mult(s, R2),
          point_neg(scalar_mult(e,curve.g))));
if P_guess1==P:
    P_guess = P_guess1
elif P_guess2==P:
    P_guess = P_guess2
print('Verification:', verify_signature(P_guess, msg, (r,s)))
```

```
(2) extracting public key from signature & verify the signature:
Private key: 0xd8c898b171a391bfa798332c177d87da50b1677a99a613ed71eeaa5c818cf43e
Public key: (0xdac667999f3d80cd0ebd70984828f67be0c2e53bedfea087849bb852f6b339af, 0x7b604f317ee28fa3be8913d56962fed13507d086116558761eff3eb20fb14213)
Message: b'How u doing?'
Signature: (0xc2d721bddb7759ebc8221df481e278d90d9fc131c7677b90746b70f5213bcb60, 0x708f96162ae26ee30e63228ca92e9993bacc627c2c88020693491395e1c09e8c)

-------------Extracting public key from signature---------------
Public key: (0xdac667999f3d80cd0ebd70984828f67be0c2e53bedfea087849bb852f6b339af, 0x7b604f317ee28fa3be8913d56962fed13507d086116558761eff3eb20fb14213)
Verification: signature matches
```

图 14: PoC impl of extracting public key from signature & verify the signature

3.3 PoC impl of Schnorr batch verification

Schnorr签名性质: $sG = (k+ed)G = kG + edG = R + eP$

Batch verification equation: $(\sum_{i=1}^{n} s_i) * G = (\sum_{i=1}^{n} R_i) + (\sum_{i=1}^{n} e_i * P_i)$. 但是这种验证方式，攻击者可以伪造签名：

攻击者私钥x_1,公钥 $\mathrm{P}_1 = x_1 * G$, 攻击者伪造公钥为$\mathrm{P}_2 = x_2 * G$的签名(攻击者不知道私钥 x_2)

(1)攻击者随机选择 $r_2, s_2, R_2 = r_2 * G$, 计算 $e_2 = h\left(P_2 \parallel R_2 \parallel m_2\right)$

(2)设置$R_1 = -\left(e_2 * P_2\right)$, 并计算 $e_1 = h\left(P_1 \parallel R_1 \parallel m\right)$

(3)得到 $s_1 = r_2 + e_1 x_1 - s_2 \bmod p$

(4)伪造的签名 $\left(R_1, s_1\right), \left(R_2, s_2\right)$ 可以通过验证： $\left(s_1 + s_2\right) * G = R_1 + R_2 + e_1 P_1 + e_2 P_2$

```
m1 = b'SHANDONG'
m2 = b'QINGDAO'
#攻击者私钥 x1 和公钥 P1
x1 = random.randrange(1, MyECDSA.curve.n)
P1 = MyECDSA.scalar_mult(x1, MyECDSA.curve.g)

#攻击者伪造公钥为 P2 的签名, 攻击者不知道私钥x2
P2 = MyECDSA.scalar_mult(random.randrange(1, MyECDSA.curve.n), MyECDSA.curve.g)

#攻击者随机选择 r2 , s2 ; 并计算e2
r2 = random.randint(1, MyECDSA.curve.n);s2 = random.randint(1, MyECDSA.curve.n)
R2 = MyECDSA.scalar_mult(r2, MyECDSA.curve.g)
#e2=h(P2||R2||m2)
e2 = MyECDSA.hash_message((hex(P2[0])[2:]+hex(P2[1])[2:]).encode('utf-8') +
                          (hex(R2[0])[2:]+hex(R2[1])[2:]).encode('utf-8') +
                            m2)

#攻击者设置特定的 R1 ; 并计算e1 , s1
R1 = MyECDSA.scalar_mult(-e2, P2)
e1 = MyECDSA.hash_message((hex(P1[0])[2:]+hex(P1[1])[2:]).encode('utf-8') +
                          (hex(R1[0])[2:]+hex(R1[1])[2:]).encode('utf-8') +
                            m1)
s1 = (r2+e1*x1-s2) % MyECDSA.curve.n

#(R1,s1)和(R2,s2)可以通过验证
e1xP1=MyECDSA.scalar_mult(e1,P1)
e2xP2=MyECDSA.scalar_mult(e2,P2)
if MyECDSA.scalar_mult(s1+s2, MyECDSA.curve.g) ==
    MyECDSA.point_add(R1,MyECDSA.point_add(R2,MyECDSA.point_add(e1xP1,e2xP2))):
   print('Success!')
```

```
(3) Schnorr Signature - Batch Verification:
Success!
```

图 15: PoC impl of Schnorr batch verification

参考文献

[1] https://blog.csdn.net/weixin_43137080/article/details/115653424

[2] https://andrea.corbellini.name/2015/05/30/elliptic-curve-cryptography-ecdh-and-ecdsa/

[3] SEC 2: Recommended Elliptic Curve Domain Parameteres, Version 2.0

A merkle_tree.py

```python
import copy
import hashlib

def hash_leaf(data,hash_function ='sha256'):# merkle 树叶节点
  hash_function =getattr(hashlib, hash_function)
  data =b'\x00'+data.encode('utf-8')
  return hash_function(data).hexdigest()

def hash_node(data,hash_function ='sha256'):# merkle 树其它节点
  hash_function =getattr(hashlib, hash_function)
  data =b'\x01'+data.encode('utf-8')
  return hash_function(data).hexdigest()

def Show_Merkle_Tree(merkle_tree,h):
  print('MerkleTree 的高度:',h)
  print('MerkleTree 从底至上每层节点哈希值:')
  for i in range(h+1):
  print('第{0}层:\n{1}\n'.format(i+1,",\n".join(merkle_tree[i])))
  print()

lst = ['a','b','c','d','e','f','g']
#lst = ['a','b','c','d','e','f']
#100k 大小测试集
'''lst = []
for i in range(100000):
lst.append(str(i))'''

def Create_Merkle_Tree(lst,hash_function ='sha256'):
  lst_hash =[]
  for i in lst:
    lst_hash.append(hash_leaf(i))
  #print("lst_hash done")
  #print('lst_hash:',lst_hash)
  merkle_tree =[copy.deepcopy(lst_hash)]#用多维列表表示 mekle 树

  if len(lst_hash)<2:print("no tracnsactions to be hashed");return 0
  h = 0 #树高度merkle
  while len(lst_hash) >1:
    h += 1
    if len(lst_hash)%2 ==0:#偶数节点
      v = []
      while len(lst_hash) >1 :
        a = lst_hash.pop(0)
        b = lst_hash.pop(0)
        v.append(hash_node(a+b, hash_function))
      #print('\nv:',v);print('len(v):',len(v))
      merkle_tree.append(v[:])#merkle 树更新一层; [:] 切片深复制效果
      #print('merkle_tree:',merkle_tree)
      lst_hash =v
    else:#奇数节点
      v = []
      last_node =lst_hash.pop(-1)
      while len(lst_hash) >1 :
        a = lst_hash.pop(0)
```

```
        b = lst_hash.pop(0)
        v.append(hash_node(a+b, hash_function))
      v.append(last_node)
      #print('v:',v);print('len(v):',len(v))
      merkle_tree.append(v[:])#merkle 树更新一层
      #print('merkle_tree:',merkle_tree)
      lst_hash =v
  return merkle_tree,h

#构造第个叶子节点存在性和验证n
def Audit_Proof(merkle_tree,h,n,leaf,hash_function ='sha256'):#为树高度, 为查找的序号hMerklen
  if n>=len(merkle_tree[0]):print("节点序号有误! ");return 0
  print("序号:{0}, 字符:{1}\查找路径n:".format(n,leaf))
  j=0 #第层j最底层需要计算叶子节点哈希值,
  L = len(merkle_tree[0])
  if L%2 ==1 and L-1==n:#叶节点为奇数个, 且 n 为最后一个节点
    hash_value =hash_leaf(leaf)
    print('第{0}层值Hash:{1}'.format(j+1,hash_value))
  elif n%2==1:
    hash_value =hash_node(merkle_tree[0][n-1]+hash_leaf(leaf),hash_function)
    print('第{0}层查询值:{1}, \n\生成的值tHash:{2}'.format(j+1,merkle_tree[0][n-1],hash_value))
  elif n%2==0:
    hash_value =hash_node(hash_leaf(leaf)+merkle_tree[0][n+1],hash_function)
    print('第{0}层查询值:{1}, \n\生成的值tHash:{2}'.format(j+1,merkle_tree[0][n+1],hash_value))
  n = n//2
  j += 1
  while j<h:#查询兄弟节点哈希值, 生成新哈希值
    L = len(merkle_tree[j])
    if L%2 ==1 and L-1==n:#节点为奇数个, 且 n 为最后一个节点
      print('第{0}层值Hash:{1}'.format(j+1,hash_value))
    elif n%2==1:
      hash_value =hash_node(merkle_tree[j][n-1]+hash_value,hash_function)
      print('第{0}层查询值:{1}, \n\t 生成的值Hash:{2}'.format(j+1,merkle_tree[j][n-1],hash_value))
    elif n%2==0:
      hash_value =hash_node(hash_value+merkle_tree[j][n+1],hash_function)
      print('第{0}层查询值:{1}, \n\t 生成的值Hash:{2}'.format(j+1,merkle_tree[j][n+1],hash_value))
    n = n//2
    j += 1

  #print(hash_value)
  print('\根节点哈希值n:',merkle_tree[h][0])
  if hash_value==merkle_tree[h][0]:
    print("节点%s 在树中Merkle"%leaf)
  else:
    print("节点%s 不在树中Merkle"%leaf)

merkle_tree,h =Create_Merkle_Tree(lst)

print('节点:',', '.join(lst))
Show_Merkle_Tree(merkle_tree,h)

#leaf = input要查找的节点: ('')
#p = int(input节点序号: ''))
#Audit_Proof(merkle_tree,h,p,leaf)
Show_Merkle_Tree(merkle_tree,h)
Audit_Proof(merkle_tree,h,4,'e')
```

B MyECDSA.py

```python
import collections
import hashlib
import random

global k_leak

EllipticCurve =collections.namedtuple('EllipticCurve', 'name p a b g n h')
curve =EllipticCurve(
'secp256k1',
# Field characteristic.
p=0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f,
# Curve coefficients.
a=0,
b=7,
# Base point.
g=(0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798,
0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8),
# Subgroup order.
n=0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141,
# Subgroup cofactor.
h=1,
)

# Modular arithmetic ############################################################
def inverse_mod(k, p):
  """Returns the inverse of k modulo p.
  This function returns the only integer x such that (x * k) % p == 1.
  k must be non-zero and p must be a prime.
  """
  if k ==0:
    raise ZeroDivisionError('division by zero')

  if k <0:
    # k ** -1 = p - (-k) ** -1 (mod p)
    return p -inverse_mod(-k, p)

  # 扩展欧几里得算法求模逆
  s, old_s =0, 1
  t, old_t =1, 0
  r, old_r =p, k

  while r !=0:
    quotient =old_r //r
    old_r, r =r, old_r -quotient *r
    old_s, s =s, old_s -quotient *s
    old_t, t =t, old_t -quotient *t

  gcd, x, y =old_r, old_s, old_t

  assert gcd ==1
  assert (k *x) % p ==1

  return x % p

```

```
55
56  # 椭圆曲线上运算 ########################################
57  def is_on_curve(point):
58    """Returns True if the given point lies on the elliptic curve."""
59    if point is None:
60      # None represents the point at infinity.
61      return True
62
63    x, y =point
64      return (y *y - x * x * x - curve.a *x - curve.b) % curve.p ==0
65
66  def point_neg(point):
67    """Returns -point."""
68    assert is_on_curve(point)
69
70    if point is None:
71      # -0 = 0
72      return None
73
74    x, y =point
75    result =(x, -y % curve.p)
76
77    assert is_on_curve(result)
78    return result
79
80
81  def point_add(point1, point2):#点加
82    """Returns the result of point1 + point2 according to the group law."""
83    assert is_on_curve(point1)
84    assert is_on_curve(point2)
85
86    if point1 is None:
87      # 0 + point2 = point2
88      return point2
89    if point2 is None:
90      # point1 + 0 = point1
91      return point1
92
93    x1, y1 =point1
94    x2, y2 =point2
95
96    if x1 ==x2 and y1 !=y2:
97      # point1 + (-point1) = 0
98      return None
99
100   if x1 ==x2:
101     # This is the case point1 == point2.
102     m = (3 * x1 * x1 + curve.a) *inverse_mod(2 *y1, curve.p)
103   else:
104     # This is the case point1 != point2.
105     m = (y1 - y2) * inverse_mod(x1 -x2, curve.p)
106
107   x3 =m * m - x1 - x2
108   y3 =y1 +m * (x3 - x1)
109   result =(x3 % curve.p,
110       -y3 % curve.p)
111
```

```
    assert is_on_curve(result)
    return result


def scalar_mult(k, point):#多倍点标量乘()
    """Returns k * point computed using the double and point_add algorithm."""
    assert is_on_curve(point)

    if k % curve.n ==0 or point is None:
        return None
    if k <0:
        # k * point = -k * (-point)
        return scalar_mult(-k, point_neg(point))

    result =None
    addend =point

    while k:
        if k & 1:
        # Add.
            result =point_add(result, addend)

        # Double.
        addend =point_add(addend, addend)
        k >>=1

    assert is_on_curve(result)
    return result

# Keypair generation and ECDSA ##################################################
def make_keypair():
    """Generates a random private-public key pair."""
    private_key =random.randrange(1, curve.n)  #私钥d
    public_key =scalar_mult(private_key, curve.g) #公钥P=dG
    return private_key, public_key


def hash_message(message):
    message_hash =hashlib.sha256(message).digest()
    e = int.from_bytes(message_hash, 'big')
    return e

    #Returns the truncated SHA512 hash of the message.
    #FIPS 180 says that when a hash needs to be truncated, the rightmost bits should be discarded.
    #当需要截断时，应丢弃最右边的位。hash
    '''message_hash = hashlib.sha512(message).digest()
    z = e >> (e.bit_length() - curve.n.bit_length())
    assert z.bit_length() <= curve.n.bit_length()
    return z'''

def sign_message(private_key, message):
    e = hash_message(message)
    r = 0
    s = 0
    while not r or not s:
        '''global k_leak;k_leak=643735664301402781313275804402842899721647129763301639134069888427910592507060
        print(' 的值k: ',k_leak)
```

```
    R_x, R_y = scalar_mult(k_leak, curve.g);
    r = R_x % curve.n;
    s = ((e + r * private_key) * inverse_mod(k_leak, curve.n)) % curve.n'''

    k = random.randrange(1, curve.n) #为保障安全性,随机生成且不能重复使用k如果泄露.会导致泄露密钥kd
    R_x, R_y =scalar_mult(k, curve.g)
    r = R_x % curve.n
    s = ((e +r * private_key) *inverse_mod(k, curve.n)) % curve.n
  return (r, s)


def verify_signature(public_key, message, signature):
  e = hash_message(message)
  r, s =signature

  w = inverse_mod(s, curve.n)
  u1 = (e * w) % curve.n
  u2 = (r * w) % curve.n

  x, y =point_add(scalar_mult(u1, curve.g),
  scalar_mult(u2, public_key))

  if (r % curve.n) ==(x % curve.n):
    return 'signature matches'
  else:
    return 'invalid signature'
```