Range Proof with Hash Function

姜舜天

2023 年 7 月 23 日

目录

1 Range Proof with Hashchains

HashWires 的灵感来自 PayWord 协议，这是 Rivest 和 Shamir 在 1996 年提出的基于哈希链的微额支付协议。最初的想法非常简单，完全基于散列链计算。简而言之，对于我们的年龄问题来说，简答的 PayWord 方法如下：现在是 2021 年，Alice 想在没有出示身份证或驾

驶执照的情况下向 Carol 证明她至少 21 岁。正如每个密码学家可能知道的那样，Alice 出生于 1978 年，当时 RSA 论文作者首次提到“在我们的场景中，我们假设 A 和 B（也称为 Alice 和 Bob）是公钥加密系统的两个用户”。但让我们暂时假设卡罗尔不是密码学家，爱丽丝真的需要 21 岁以上的证明；而且他们都相信可信第三方颁发的证书。此外，假设我们想在 2100 年之前使用这个证明系统。所以可信第三方会为 Alice 提供一个签名后的使用两种抗碰撞哈希函数 H_0, H_1 密码学承诺

1.1 Trusted issuer

Algorithm 1 Trusted issuer

1. 选取至少 128bit 长的随机种子 $seed$
2. 计算 $s = H_0(seed)$, 并以这个代表 1978 年 (Alice 的出生年份)
3. 计算 $k = 2100 - 1978 = 122$ 作为从 1978 到最大支持年份的距离
4. 计算承诺 $c = H_1^k(s)$，其中 k 代表 H_1 函数的迭代次数
5. 输出 s 和在 c 上做的签名 sig_c

上述算法核心代码如下：

Listing 1: Trusted_Issuler

```
def Trusted_Issuer(born):
    sk, pk = SM2Sign.SM2_Key_Generate()
    seed = random.randint(2 ** 128, 2 ** 129)
    seed = str(seed).encode()
    s = hashlib.sha512(seed)
    k = 2100 - born
    c = s
    for i in range(k):
        c = hashlib.sha256(c.hexdigest().encode())
    sigc = SM2Sign.sm2_sig(c.hexdigest(), sk)
    return s, sigc, pk
```

1.2 Alice

Algorithm 2 Alice

1. 计算 $d_0 = 2000 - 1978$
2. 计算并输出证据 $p = H_1^{d_0}(s)$
3. 将 (p, sig_c) 给 Carol

上述算法核心代码如下:

Listing 2: Alice

```
def Prover(born, s, sigc):
    d_0 = 2000 - born
    S = int(s, 16)
    p_1 = hashlib.sha256(s.encode())
    for i in range(d_0-1):
        p_1 = hashlib.sha256(p_1.hexdigest().encode())
    return  p_1
```

1.3 Carol

Algorithm 3 Carol

1. 计算 $d_1 = 2100 - 2000$
2. 计算承诺 $c = p^{d_1}$, 该式一定成立因为 $100 + 22 = 122$
3. 使用可信第三方提供的公钥验证签名是否对计算出的 c 成立

上述算法核心代码如下:

Listing 3: Carol

```
def Verifier(p_1,sigc,pk):
    d_1 = 2100 - 2000
    c_ = hashlib.sha256(p_1.encode())
    for i in range(d_1-1):
        c_ = hashlib.sha256(c_.hexdigest().encode())
    sigc_ = SM2Sign.sm2_ver(c_.hexdigest(),sigc[0],sigc[1],pk)
    print(sigc_)
```

2 具体实现

按照 CS 模式实现，由可信第三方担任服务器，通过多线程编程来实现多方连接，证明方和验证方通过服务器进行通信，具体实现时使用的两个散列函数 H_0, H_1 分别为 SHA-512 与 SHA-256, 通过 for 循环执行迭代

代码过长，请见附录

服务器代码中 handle_client 负责处理另外两方，而 judge 函数负责解析发来的信息并将完成对应指令的任务，具体操作如下

操作序号	Alice 对应操作功能	Carol 对应操作功能
0	退出客户端	退出客户端
1	输入出生年份并发给服务器进行秘密、公钥、签名的计算	向服务器请求公钥和签名
2	计算 $H_1^p(s)$ 并发给服务器，由服务器转发给 Carol	向服务器请求 Alice 计算的结果进而进行下一步计算
3	——	本地执行验证计算

3 实现效果

测试代码请见附录

3.1 测试流程

测试流程如下：

1. 运行服务器 Trusted.py，此时会提示“服务器启动，等待客户端连接...”。

2. 运行 Alice 端 Prover.py，此时服务器会提示“客户端（xxx，xxx，xxx，xxx）连接成功”，同时客户端会提示“请输入你下一步的操作”，这里键盘输入‘1’回车会立刻提示“申请验证，请输入你的出生年份”，继续输入”1978“，会显示当前的发送的指令“Prover11978”，这条指令会由服务器解析，服务器相继提示“消息：Prover11978 “”Prover 执行的操作为：1”，然后进行计算得到结果，提示：“生成秘密：xxx”，“生成公钥：xxx”，”生成签名：xxx“，最后将要发送给 Alice 的序列输出并把序列发送给 Alice，Alice 会进行解析，提示：“收到秘密为：xxx“，”收到公钥为：（xxx，xxx）“，”收到签名为：[xxx，xxx]”，然后提示“请输入你下一步的操作”。

3. 接着输入‘2’，提示当前的发送的指令“Prover2”，接着提示“计算 Hash：”并进行计算，计算完后显示得到的 hash 结果和发送给服务器的序列。Alice 端任务到此结束。

4. 运行 Carol 端 Verifier.py，服务器会提示“客户端（xxx，xxx，xxx，xxx）连接成功”，同时客户端会提示“请输入你下一步的操作”，键盘输入‘1’并回车，提示“请求当前验证公钥和签名“和”Verifier1“，然后服务器提示“消息：Verifier1”“Verifier 执行的操作为：1”“Verifier 请求公钥签名”，Carol 端拿到公钥和签名并提示：”收到公钥为：（xxx，xxx）“，”收到签名为：[xxx，xxx]”，然后提示“请输入你下一步的操作”。

5. 接着输入‘2’，提示“请求 Hash：和当前的发送的指令“Verifier2”，服务器端提示消息：“Verifier2”，“Verifier 执行的操作为：2”，“Verifier 请求 Hash”，Carol 端收到后会将序列输出并解析，提示“收到 Hash：xxx”，然后提示“请输入你下一步的操作”。

6. 接着输入‘3’，提示“验证计算”，计算后会输出验证结果，如果成立则结果为 1，通信结束

3.2 执行结果

执行结果如下：

```
服务器启动，等待客户端连接...
客户端 ('127.0.0.1', 65301) 连接成功。
消息： Prover11978
Prover执行的操作为： 1
生成秘密： 7a15d75a26792db35e552f2cdcedb95400835bb0e8d7f523744c7799a1b5d3b89a332f56756ab9eb7dd3abfc030674b374a4e31bda637f7e0479e51987d64cb9
生成公钥： (219497334849019706187658267637044109590744926328419153500828675966465596912,722123733114043659601026756768811492272355072890285617577
生成签名： [825293897782855542528462364298627057500589612300140888736676531011552891190081027914492321902886460284851440135806599064447627420144
778252938977828555425284623642986270575005896123001408887366765310115528911900810279144923219028864602848514401358065990644476274201442316309203
消息： Prover2
Prover执行的操作为： 2
接收hash
fac0d1e82e5449acfe85da387524e5fa172d9a668df5200e5f72de61c6e2f5f1
收到Hash： 113418895506827192296228027095459493680035254885323823119722804435712152368625
客户端 ('127.0.0.1', 65305) 连接成功。
消息： Verifier1
Verifier执行的操作为： 1
Verifier请求公钥签名
778252938977828555425284623642986270575005896123001408887366765310115528911900810279144923219028864602848514401358065990644476274201442316309203
消息： Verifier2
Verifier执行的操作为： 2
Verifier请求Hash
```

图 1: 服务器执行结果

```
请输入你下一步的操作1
申请验证，请输入你的出生年份1978
Prover11978
收到秘密为： 7a15d75a26792db35e552f2cdcedb95400835bb0e8d7f523744c7799a1b5d3b89a332f56756ab9eb7dd3abfc030674b374a4e31bda637f7e0479e51987d64cb9
收到公钥为： (219497334849019706187658267637044109590744926328419153500828675966465596912,722123733114043659601026756768811492272355072890285617577
收到签名为： [2938977828555425284623642, 986270575005896123001408887366765310115528911900810279144923219028864602848514401358065990644476274201442
请输入你下一步的操作2
Prover2
计算Hash:
fac0d1e82e5449acfe85da387524e5fa172d9a668df5200e5f72de61c6e2f5f1
fac0d1e82e5449acfe85da387524e5fa172d9a668df5200e5f72de61c6e2f5f1
请输入你下一步的操作Traceback (most recent call last):
  File "/Users/mac/Documents/Py Programs/RangeProof/Prover.py", line 35, in <module>
    op = input('请输入你下一步的操作')
KeyboardInterrupt
```

图 2: Alice 端执行结果

```
请输入你下一步的操作1
请求当前验证公钥和签名
Verifier1
收到公钥为： (219497334849019706187658267637044109590744926328419153500828675966465596912,722123733114043659601026756768811492272355072890285617577
收到签名为： [8252938977828555425284623642986270575005896123001408887366765310115528911900 8, 1027914492321902886460284851440135806599064447627420144
请输入你下一步的操作2
请求hash
Verifier2
fac0d1e82e5449acfe85da387524e5fa172d9a668df5200e5f72de61c6e2f5f1
收到Hash： fac0d1e82e5449acfe85da387524e5fa172d9a668df5200e5f72de61c6e2f5f1
请输入你下一步的操作3
验证计算
1
请输入你下一步的操作Traceback (most recent call last):
  File "/Users/mac/Documents/Py Programs/RangeProof/Verifier.py", line 28, in <module>
    ov = input('请输入你下一步的操作')
KeyboardInterrupt
```

图 3: Carol 端执行结果

4 总结

上述工作是因为 Carol 确信 Alice 有一些发布的秘密，至少有 100 个哈希链节点长，这反过来意味着 Alice 出生在 2000 年或之前（否则发行人永远不会为爱丽丝提供这样的长链）。此外，证明实际上是单个哈希值，只有 32 字节。尽管它很简单，但上述解决方案有一个重大的缺点：承诺、证明生成和最终证明验证的成本在最坏的情况下是与 k 成线性关系的。

请注意，如果上述 21 岁以上示例中的时间粒度是分钟而不是年，我们期望 k 的大小在数百万范围内，这本质上需要数百万个哈希调用。因此，使用 PayWord 进行范围证明实际上只适用于小域，其对大范围（即 32 位或 64 位数）的性能并不实用。

参考文献

[1] https://zkproof.org/2021/05/05/hashwires-range-proofs-from-hash-functions/

A 附录

A.1 服务器

Listing 4: Trusted

```
def handle_client(client_socket):
    while True:
        try:
            data = client_socket.recv(1024)
            if not data:
                break
            # 处理接收到的数据
            judge(data, client_socket)
        except:
            break

    # 关闭客户端连接
    client_socket.close()

def judge(message,socket):

    global pk,s,sigc,hash,hash_seq
    print('消息: ',message.decode())
    P = message.decode().find('Prover')
    V = message.decode().find('Verifier')

    if P != -1:
        op = ''
        op = message.decode()[P+6]
        print('Prover执行的操作为: ',op)
```

```
26          if op == '1':
27              born = int(message.decode()[P+7:])
28              s , sigc , pk = Trusted_Issuer(born)
29
30              print('生成秘密: ',s.hexdigest())
31              clients[0].send(s.hexdigest().encode())
32              print('生成公钥: ',pk)
33
34              serialized_point = pickle.dumps(pk)
35              clients[0].send(serialized_point)
36              print('生成签名: ',sigc)
37
38              seq = str(sigc[0]) + str(sigc[1])
39              seq = str(len(str(sigc[0]))) + seq
40              print(seq)
41              clients[0].send(seq.encode())
42          if op == '2':
43              print('接收hash')
44              hash_seq = clients[0].recv(1024).decode()
45              print(hash_seq)
46              hash = int(hash_seq,16)
47              print('收到Hash: ',hash)
48
49      elif V!= -1:
50          ov = ''
51          ov = message.decode()[V+8]
52          print('Verifier执行的操作为: ',ov)
53          if ov=='1':
54              print('Verifier请求公钥签名')
55              serialized_point = pickle.dumps(pk)
56              clients[1].send(serialized_point)
57
58              seq = str(sigc[0]) + str(sigc[1])
59              seq = str(len(str(sigc[0]))) + seq
```

```
60                print(seq)
61                clients[1].send(seq.encode())
62            if ov=='2':
63                print('Verifier请求Hash')
64                clients[1].send(hash_seq.encode())
65
66        return
67
68
69    # 创建TCP套接字
70    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
71
72    # 绑定服务器的地址和端口
73    server_address = ('', 8888)
74    server_socket.bind(server_address)
75
76    # 监听连接
77    server_socket.listen(2)
78    print('服务器启动, 等待客户端连接...')
79    clients = []
80    host = socket.gethostname()
81    while True:
82        try:
83            # 接受客户端连接
84            client_socket, client_address = server_socket.accept()
85            print(f'客户端 {client_address} 连接成功。')
86            # 将客户端加入列表
87            clients.append(client_socket)
88
89            # 启动线程处理客户端连接
90            t = threading.Thread(target=handle_client, args=(
                  client_socket,))
91            t.start()
92        except KeyboardInterrupt:
```

```
93            break
94
95  # 关闭服务器套接字
96  server_socket.close()
```

A.2 Prover

Listing 5: Prover

```
1
2   s = socket.socket()
3   host = socket.gethostname()
4   port = 8888
5   s.connect((host, port))
6   # print(s.recv(10).decode())
7   pk = 0
8   sig = 0
9   secret = 0
10  id = 'Prover'
11  born = 0
12  p_0, p_1 = 0, 0
13  # s.send(id.encode())
14  # s.send(str(born).encode())
15  while 1:
16      op = input('请输入你下一步的操作')
17      if op == '0':
18          print('连接结束')
19          break
20      if op == '1':
21          born = input('申请验证, 请输入你的出生年份')
22          seq = (id + op + born)
23          print(seq)
24          s.send(seq.encode())
25          # sigc = s.recv(1024)
```

```
26          # print(sigc)
27
28          secret = s.recv(1024).decode()
29          print('收到秘密为: ', secret)
30          pk = s.recv(363)
31          # print(pk)
32          pk = pickle.loads(pk)
33          print('收到公钥为: ', pk)
34
35          seq = s.recv(160).decode()
36          # print(seq)
37          leng = int(seq[:2])
38          # print(len)
39          sig = [int(seq[2:2 + leng]), int(seq[2 + leng:])]
40          print('收到签名为: ', sig)
41
42      if op == '2':
43          seq = (id + op)
44          print(seq)
45          s.send(seq.encode())
46          p_1 = Prover(int(born), secret, sig)
47          print('计算Hash: ')
48          print(p_1.hexdigest())
49          hash_seq = p_1.hexdigest()
50          print(hash_seq)
51          s.send(hash_seq.encode())
52
53  s.close()
```

A.3 Verifier

Listing 6: Verifier

```
1           s = socket.socket()
```

```
host = socket.gethostname()
port = 8888
s.connect((host, port))
pk = 0
sig = 0
id = 'Verifier'
hash = 0
while 1:
    ov = input('请输入你下一步的操作')
    if ov == '0':
        print('连接结束')
        break
    if ov == '1':

        print('请求当前验证公钥和签名')
        seq = (id + ov)
        print(seq)
        s.send(seq.encode())
        pk = s.recv(363)
        pk = pickle.loads(pk)
        print('收到公钥为: ', pk)

        seq = s.recv(170).decode()
        leng = int(seq[:2])
        sig = [int(seq[2:2 + leng]), int(seq[2 + leng:])]
        print('收到签名为: ',sig)
    if ov =='2':
        print('请求hash')
        seq = (id + ov)
        print(seq)
        s.send(seq.encode())
        hash_seq = s.recv(1024).decode()
        print(hash_seq)
        hash = hash_seq
```

```
36            print('收到Hash: ', hash)
37
38        if ov =='3':
39            print('验证计算')
40            Verifier(hash, sig ,pk)
```