Project06——Google Password Checkup 验证

本项目复现论文《On Deploying Secure Computing: Private Intersection-Sum-with-Cardinality》提出的私有交集求和协议(Private Intersection-Sum Protocol)，该协议允许两方在不泄露各自私有数据集的情况下，计算它们数据集的交集大小以及较集中元素的关联值之和。

1 协议实现

协议使用了以下原语：DDH假设成立的群$\mathcal{G}$，加法同态加密方案$(AGen，AEnc，ADec)$(该项目使用Paillier加法同态加密方案)，随机寓言机模型H。

具体协议如下：

- 输入：

双方：素数阶群$\mathcal{G}$，标识符空间$\mathcal{U}$，哈希函数$H:\mathcal{U}\rightarrow\mathcal{G}$。

P_1：集合$V=\{v_i\}_{i=1}^{m_1},v_i\in\mathcal{U}$。

P_2：集合$W=\{(w_i,t_i)\}_{i=1}^{m_2},w_i\in\mathcal{U},t_i\in\mathbb{Z}^+$。

- 初始化：

双方生成私钥k_1,k_2。

P_2生成加法同态加密密钥对$(pk,sk)\leftarrow AGen(\lambda)$，并将$pk$发送给$P_1$。

- 第一轮

(1) $\forall i=1,2,\cdots m_1,P_1$计算$H(v_i)^{k_1}$；

(2) P_1将洗牌后的$\{H(v_i)^{k_1}\}_{i=1}^{m_1}$发送给$P_2$。

具体实现如下：

```
    def party1_round1(self, V: List[str]) -> Tuple[List[int], int]:
        """P1第1轮: 哈希并指数化自己的元素"""
        self.k1 = random.randint(1, self.group.p - 1)
        hashed_exponents = []
        for v in V:
            h = self.group.hash_to_group(v)
            hashed_exponents.append(self.group.exponentiate(h, self.k1))

        # 打乱顺序
        random.shuffle(hashed_exponents)
        return hashed_exponents, self.k1
```

- 第二轮

(1) P_2对收到的每个值计算$H(v_i)^{k_1k_2}$。

(2) P_2将洗牌后的$Z=\{H(v_i)^{k_1k_2}\}_{i=1}^{m_1}$发送给$P_1$。

(3) P_2对每个(w_j,t_j)计算$H(w_j)^{k_2}$和$AEnc(t_j)$。

(4) P_2将洗牌后的$\{(H(w_j)^{k_2}, AEnc(t_j))\}_{j=1}^{m_2}$发送给$P_1$。

具体实现如下：

```
    def party2_round2(self, received_from_p1: List[int], W: List[Tuple[str,
int]]) -> Tuple[
        List[int], List[Tuple[int, paillier.EncryptedNumber]],
paillier.PaillierPublicKey]:
        """P2第2轮: 双重指数化并加密关联值"""
        self.k2 = random.randint(1, self.group.p - 1)
        self.ahe = AdditiveHomomorphicEncryption()

        # 处理P1的元素
        double_exponents = []
        for elem in received_from_p1:
            double_exponents.append(self.group.exponentiate(elem, self.k2))
        random.shuffle(double_exponents)

        # 处理自己的元素
        hashed_encrypted = []
        for w, t in W:
            h = self.group.hash_to_group(w)
            hashed = self.group.exponentiate(h, self.k2)
            encrypted = self.ahe.encrypt(t)
            hashed_encrypted.append((hashed, encrypted))
        random.shuffle(hashed_encrypted)

        return double_exponents, hashed_encrypted, self.ahe.public_key
```

- 第三轮

(1) P_1对收到的每个$\{(H(w_j)^{k_2}, AEnc(t_j))\}$计算$H(w_j)^{k_2k_1}$。

(2) P_1计算交集$J = \{j : H(w_j)^{k_1k_2} \in Z\}$。

(3) P_1计算交集和的密文：$AEnc(pk, S_J) = ASum(\{AEnc(t_j)\}_{j\in J}) = AEnc(\sum_{j\in J} t_j)$。并使用$ARefresh$随机化密文，将其发送给$P_2$。

具体实现如下：

```
    def party1_round3(self, received_double_exponents: List[int],
                      received_hashed_encrypted: List[Tuple[int,
paillier.EncryptedNumber]],
                      k1: int, V: List[str]) -> paillier.EncryptedNumber:
        """P1第3轮: 计算交集和"""
        # 计算交集
        intersection_indices = []
        encrypted_sum = None

        # 将P1的双重指数化结果存入集合便于查找
        p1_elements = set(received_double_exponents)

        for idx, (hashed, encrypted) in enumerate(received_hashed_encrypted):
            # 完成双重指数化
            double_hashed = self.group.exponentiate(hashed, k1)
```

```
            # 检查是否在交集中
            if double_hashed in p1_elements:
                intersection_indices.append(idx)

                # 同态累加
                if encrypted_sum is None:
                    encrypted_sum = encrypted
                else:
                    encrypted_sum = self.ahe.add(encrypted_sum, encrypted)

        # 随机化最终的和（Paillier加密本身已经具有随机性）
        return encrypted_sum
```

- 输出：

P_2解密得到交集和S_J。

具体实现如下：

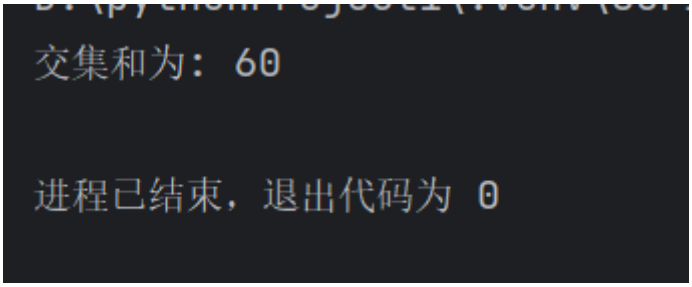
```
    def party2_output(self, encrypted_sum: paillier.EncryptedNumber) -> int:
        """P2输出：解密得到交集和"""
        return self.ahe.decrypt(encrypted_sum)
```

2 结果展示

测试用例：

```
def test_protocol():
    # 模拟数据
    V = ["user1", "user2", "user3", "user4"]
    W = [("user2", 10), ("user3", 20), ("user4", 30), ("user6", 40)]

    protocol = DDHPrivateIntersectionSum()

    # P1第1轮
    p1_round1_result, k1 = protocol.party1_round1(V)

    # P2第2轮
    p2_round2_result1, p2_round2_result2, p2_pubkey =
protocol.party2_round2(p1_round1_result, W)

    # P1第3轮
    p1_round3_result = protocol.party1_round3(p2_round2_result1,
p2_round2_result2, k1, V)

    # P2输出
    intersection_sum = protocol.party2_output(p1_round3_result)

    print(f"交集和为：{intersection_sum}")  # 应输出60 (user2 + user3 + user4)
```

运行结果如下：



可以看到协议正确运行。