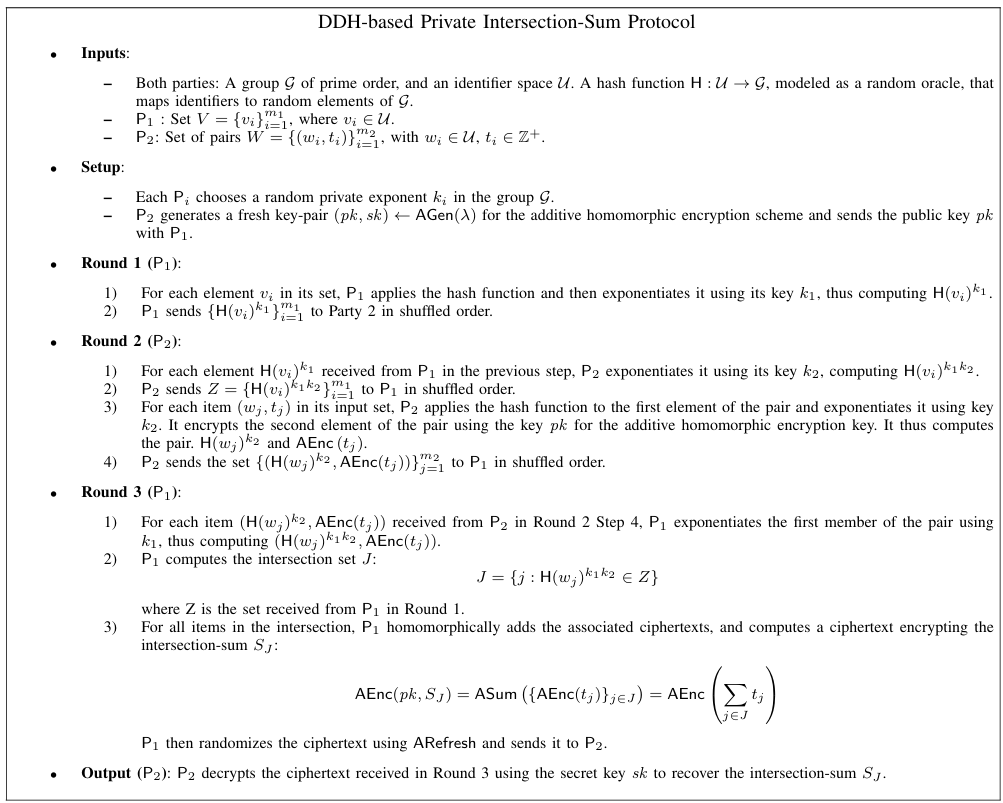
Project6

密码22.1 张靖陶 202200460011

1. 算法结构



## Inputs（输入）

**双方与公共参数**：  
 一个素数阶群 G与一个标识符空间U。存在一个将标识符映射到G的哈希函数 H:U→G，把标识看成随机预言机输出的群元素。

**P1 的集合**：

**P2 的集合**：

## Setup（初始化）

每个参与方Pi在群G中独立选择一个私钥ki。P2为“可加同态加密”方案生成密钥对 (pk,sk)←AGen (λ)，并把公钥pk发送给P1。

## Round 1（第 1 轮，P1）

对其集合中每个元素vi，P1先做哈希再用密钥k1幂运算：计算

P1将打乱顺序后的多集合 发送给P2。

## Round 2（第 2 轮，P2）

对收到的每个，P2再用自己的密钥 k2幂运算，得到 

P2将打乱后的集合 发送给P1。

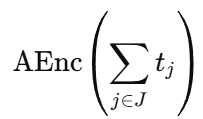
对自己输入集合中的每一对 (wj,tj)，P2计算哈希并用k2幂运算，得到第一分量 ；并用公钥pk加密第二分量得到AEnc(tj)。于是得到配对 P2将打乱后的配对集合发送给P1。

## Round 3（第 3 轮，P1）

对每个收到的配对 P1对第一分量再用k1幂运算，得到:

P1用第2轮收到的集合Z做匹配： 即J正是交集对应的索引集合（但顺序被打乱，且P1不知道这些索引对应的原始标识符）。

P1对{ AEnc(tj)∣j∈J }做**同态加法**，得到一个密文，它加密的就是交集元素对应的t值之和：



然后P1随机化该密文并将其发给P2。

## Output（输出，P2）

P2用私钥sk解密上一步的密文，得到交集求和结果

1. Python实现

|  |
| --- |
| import random |
| import hashlib |
| import math |
| from dataclasses import dataclass |
| from typing import List, Tuple |
|  |
| # 参数 |
| RFC3526\_MODP\_2048\_P = int( |
| "FFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD1" |
| "29024E088A67CC74020BBEA63B139B22514A08798E3404DD" |
| "EF9519B3CD3A431B302B0A6DF25F14374FE1356D6D51C245" |
| "E485B576625E7EC6F44C42E9A637ED6B0BFF5CB6F406B7ED" |
| "EE386BFB5A899FA5AE9F24117C4B1FE649286651ECE45B3D" |
| "C2007CB8A163BF0598DA48361C55D39A69163FA8FD24CF5F" |
| "83655D23DCA3AD961C62F356208552BB9ED529077096966D" |
| "670C354E4ABC9804F1746C08CA18217C32905E462E36CE3B" |
| "E39E772C180E86039B2783A2EC07A28FB5C55DF06F4C52C9" |
| "DE2BCBF6955817183995497CEA956AE515D2261898FA0510" |
| "15728E5A8AACAA68FFFFFFFFFFFFFFFF", 16) |
| RFC3526\_MODP\_2048\_G = 2 |
| RFC3526\_MODP\_2048\_Q = (RFC3526\_MODP\_2048\_P - 1) // 2 #安全素数 q |
|  |
| def sha256\_int(data: bytes) -> int: |
| return int.from\_bytes(hashlib.sha256(data).digest(), "big") |
|  |
| def hash\_to\_group(u: str) -> int: |
| x = sha256\_int(u.encode("utf-8")) % RFC3526\_MODP\_2048\_Q |
| if x == 0: |
| x = 1 |
| return pow(RFC3526\_MODP\_2048\_G, x, RFC3526\_MODP\_2048\_P) |
|  |
| # 简单 Paillier 实现 |
| def \_is\_probable\_prime(n: int, k: int = 16) -> bool: |
| if n < 2: |
| return False |
| small\_primes = [2,3,5,7,11,13,17,19,23,29] |
| for p in small\_primes: |
| if n % p == 0: |
| return n == p |
| d = n - 1 |
| s = 0 |
| while d % 2 == 0: |
| s += 1 |
| d //= 2 |
| for \_ in range(k): |
| a = random.randrange(2, n - 2) |
| x = pow(a, d, n) |
| if x == 1 or x == n - 1: |
| continue |
| composite = True |
| for \_ in range(s - 1): |
| x = pow(x, 2, n) |
| if x == n - 1: |
| composite = False |
| break |
| if composite: |
| return False |
| return True |
|  |
| def \_rand\_prime(bits: int) -> int: |
| while True: |
| cand = random.getrandbits(bits) | 1 | (1 << (bits - 1)) |
| if \_is\_probable\_prime(cand): |
| return cand |
|  |
| @dataclass |
| class PaillierPublicKey: |
| n: int |
| n2: int |
| g: int |
|  |
| @dataclass |
| class PaillierSecretKey: |
| lam: int |
| mu: int |
| n: int |
| n2: int |
| g: int |
|  |
| # 生成 Paillier 密钥对 |
| def paillier\_keygen(bits: int = 512) -> Tuple[PaillierPublicKey, PaillierSecretKey]: |
| p = \_rand\_prime(bits // 2) |
| q = \_rand\_prime(bits // 2) |
| while q == p: |
| q = \_rand\_prime(bits // 2) |
| n = p \* q |
| n2 = n \* n |
| lam = (p - 1) \* (q - 1) // math.gcd(p - 1, q - 1) |
| g = n + 1 |
| def L(u): return (u - 1) // n |
| mu = pow(L(pow(g, lam, n2)), -1, n) |
| pk = PaillierPublicKey(n=n, n2=n2, g=g) |
| sk = PaillierSecretKey(lam=lam, mu=mu, n=n, n2=n2, g=g) |
| return pk, sk |
|  |
| def paillier\_enc(pk: PaillierPublicKey, m: int) -> int: |
| m = m % pk.n |
| while True: |
| r = random.randrange(1, pk.n) |
| if math.gcd(r, pk.n) == 1: |
| break |
| c = (pow(pk.g, m, pk.n2) \* pow(r, pk.n, pk.n2)) % pk.n2 |
| return c |
|  |
| def paillier\_dec(sk: PaillierSecretKey, c: int) -> int: |
| def L(u, n): return (u - 1) // n |
| u = pow(c, sk.lam, sk.n2) |
| m = (L(u, sk.n) \* sk.mu) % sk.n |
| return m |
|  |
| def paillier\_add(pk: PaillierPublicKey, c1: int, c2: int) -> int: |
| return (c1 \* c2) % pk.n2 |
|  |
| def paillier\_rerandomize(pk: PaillierPublicKey, c: int) -> int: |
| while True: |
| r = random.randrange(1, pk.n) |
| if math.gcd(r, pk.n) == 1: |
| break |
| return (c \* pow(r, pk.n, pk.n2)) % pk.n2 |
|  |
| @dataclass |
| class P1Input: |
| V: List[str] |
|  |
| @dataclass |
| class P2Input: |
| W: List[Tuple[str, int]] |
|  |
| def ddh\_pis\_protocol(P1: P1Input, P2: P2Input, paillier\_bits: int = 512, verbose: bool = True): |
|  |
| p = RFC3526\_MODP\_2048\_P |
| q = RFC3526\_MODP\_2048\_Q |
| g = RFC3526\_MODP\_2048\_G |
|  |
| k1 = random.randrange(1, q) |
| k2 = random.randrange(1, q) |
| if verbose: |
| print("初始化（Setup）") |
| print("P1 随机选取私钥 k1，P2 随机选取私钥 k2（保密）") |
| print("P2 生成 Paillier 密钥对并把公钥发送给 P1") |
|  |
| pk, sk = paillier\_keygen(bits=paillier\_bits) |
| if verbose: |
| print(f"已生成 Paillier 密钥（公钥模 n 的位长约 {pk.n.bit\_length()} 位）") |
| print() |
|  |
| R1 = [pow(hash\_to\_group(vi), k1, p) for vi in P1.V] |
| random.shuffle(R1) # 打乱顺序以防位置关联 |
| if verbose: |
| print("第 1 轮") |
| print(f"P1 对 V 中 {len(P1.V)} 个元素计算 H(v)^{k1} 并打乱后发送给 P2。") |
| print() |
|  |
| Z = [pow(x, k2, p) for x in R1] |
| random.shuffle(Z) |
| pairs = [] |
| for (wj, tj) in P2.W: |
| h\_w\_k2 = pow(hash\_to\_group(wj), k2, p) |
| enc\_tj = paillier\_enc(pk, tj) |
| pairs.append((h\_w\_k2, enc\_tj)) |
| random.shuffle(pairs) |
| if verbose: |
| print("第 2 轮") |
| print(f"P2 对收到的 R1 中每项再做 ^k2，得到 Z 并发送给 P1。") |
| print(f"P2 还发送其自身 W 转换后的配对 (H(w)^{k2}, Enc(t)) 共 {len(pairs)} 项，顺序也已打乱。") |
| print() |
|  |
| pairs\_k1 = [(pow(hk2, k1, p), ct) for (hk2, ct) in pairs] |
| Zset = set(Z) |
| J\_indices = [i for i, (h12, ct) in enumerate(pairs\_k1) if h12 in Zset] |
| if verbose: |
| print("第 3 轮") |
| print("P1 将接收到的 pairs 中的第一分量再做 ^k1，变为 H(w)^{k1 k2}，并与 Z 比较匹配。") |
| print(f"P1 识别出交集中的元素数量（|J|） = {len(J\_indices)}") |
| print() |
|  |
| if not J\_indices: |
| C\_sum = paillier\_enc(pk, 0) |
| else: |
| C\_sum = pairs\_k1[J\_indices[0]][1] |
| for idx in J\_indices[1:]: |
| C\_sum = paillier\_add(pk, C\_sum, pairs\_k1[idx][1]) |
|  |
| C\_rand = paillier\_rerandomize(pk, C\_sum) |
| if verbose: |
| print("P1 对同态求和结果进行重随机化后发送给 P2。") |
| print() |
|  |
| # P2 用私钥解密得到交集求和结果 |
| sum\_value = paillier\_dec(sk, C\_rand) |
| if verbose: |
| print("输出") |
| print(f"P2 解密得到交集元素对应 t 值之和 = {sum\_value}") |
| print() |
|  |
| expected = 0 |
| intersected = [] |
| P1set = set(P1.V) |
| for (w, t) in P2.W: |
| if w in P1set: |
| expected += t |
| intersected.append(w) |
| if verbose: |
| print(f" 交集元素（明文） = {intersected}") |
| print(f" 明文求和（期望） = {expected}") |
| if sum\_value == expected: |
| print(" 解密结果与期望一致。") |
| else: |
| print(" 解密结果与期望不一致。") |
| return { |
| "decrypted\_sum": sum\_value, |
| "expected\_sum": expected, |
| "intersection\_items": intersected, |
| "J\_size": len(J\_indices), |
| "paillier\_n\_bits": pk.n.bit\_length() |
| } |
|  |
| def main\_demo(): |
| random.seed(42) |
| P1 = P1Input(V=["alice", "bob", "carol", "dave"]) |
| P2 = P2Input(W=[("bob", 5), ("erin", 7), ("carol", 11), ("frank", 13)]) |
|  |
| print("输入：") |
| print(" P1 的集合 V =", P1.V) |
| print(" P2 的集合 W (带权重) =", P2.W) |
| print() |
|  |
| # 运行协议 |
| result = ddh\_pis\_protocol(P1, P2, paillier\_bits=512, verbose=True) |
|  |
| print() |
| print("最终摘要：") |
| print(" 解密得到的交集求和 =", result["decrypted\_sum"]) |
| print(" 明文期望的交集求和 =", result["expected\_sum"]) |
| print(" 明文交集元素列表 =", result["intersection\_items"]) |
| print(" P1识别出的交集大小 |J| =", result["J\_size"]) |
| print(" Paillier模n的位长度 =", result["paillier\_n\_bits"]) |
|  |
| if \_\_name\_\_ == "\_\_main\_\_": |
| main\_demo() |