山东大学

SHANDONG UNIVERSITY

Project 6: Google Password Checkup

姓　　名: 谢星豪

学　　号: 202100161170

专　　业: 网络空间安全

班　　级: 网安一班

2025 年 8 月 11 日

目录

1 实验内容

Project 6: Google Password Checkup 验证

来自刘巍然老师的报告 google password checkup，参考论文 https://eprint.iacr.org/2019/723.pdf 的 section 3.1 ，也即 Figure 2 中展示的协议，尝试实现该协议，（编程语言不限）。

2 协议概述

该协议是一个基于迪菲-赫尔曼问题（DDH-based）的隐私交集求和（Private Intersection-Sum）协议。协议的目标是让持有数据集的两方（P1 和 P2）在不泄露各自数据集内容的前提下，计算出他们数据集交集的对应值的总和。

具体来说，P1 持有集合 $V = \{v_i\}_{i=1}^{m_1}$，P2 持有一个由“标识-值”对组成的集合 $W = \{(w_j, t_j)\}_{j=1}^{m_2}$，其中 t_j 是与标识 w_j 相关联的整数。协议的最终目标是计算交集元素的对应值之和，即 $\sum_{v_i=w_j} t_j$，同时保护 P1 的 V 集合和 P2 的 W 集合的隐私。

协议的安全性依赖于决策迪菲-赫尔曼（DDH）假设和加法同态加密方案的安全性。

2.1 详细步骤

1. 输入 (Inputs)

- **公共参数**:
  - G: 一个素数阶的乘法循环群。
  - $\mathcal{U}$: 一个标识符空间。
  - $H : \mathcal{U} \to G$: 一个哈希函数，被建模为随机预言机（Random Oracle）。
- **参与方数据**:
  - P1 方（Party 1）持有集合 $V = \{v_i\}_{i=1}^{m_1}$，其中 $v_i \in \mathcal{U}$。
  - P2 方（Party 2）持有集合 $W = \{(w_j, t_j)\}_{j=1}^{m_2}$，其中 $w_j \in \mathcal{U}$ 并且 $t_j \in \mathbb{Z}^+$。

2. 设置 (Setup)

- 每一方都选择一个私有的随机指数作为自己的密钥。
  - P1 选择私钥 $k_1 \in \mathbb{Z}_q^*$ (其中 q 是群 G 的阶)。
  - P2 选择私钥 $k_2 \in \mathbb{Z}_q^*$。
- P2 生成一个加法同态加密方案的密钥对 $(pk, sk) \leftarrow \text{AGen}(\lambda)$，其中 λ 是安全参数。然后 P2 将公钥 pk 发送给 P1。

3. 轮次 1 (Round 1)

此轮的目的是 P1 对其集合中的元素进行盲化操作。

1. 对于其集合 V 中的每一个元素 v_i，P1 首先计算其哈希值 $H(v_i)$，然后用自己的私钥 k_1 对结果进行指数运算，得到 $H(v_i)^{k_1}$。
2. P1 将所有计算出的值 $\{H(v_i)^{k_1}\}_{i=1}^{m_1}$ 以随机顺序（shuffled order）发送给 P2。这样做是为了防止 P2 通过位置信息猜测原始的 v_i。

4. 轮次 2 (Round 2)

此轮的目的是 P2 找出交集元素，并对交集元素的关联值 t_j 进行加密。

1. P2 接收到来自 P1 的集合 $\{H(v_i)^{k_1}\}_{i=1}^{m_1}$。对于收到的每一个元素，P2 用自己的私钥 k_2 对其进行指数运算，得到 $(H(v_i)^{k_1})^{k_2} = H(v_i)^{k_1 k_2}$。
2. P2 将这个双重盲化的集合 $\{H(v_i)^{k_1 k_2}\}_{i=1}^{m_1}$ 命名为 Z，并以随机顺序发送回给 P1。
3. 与此同时，P2 对自己的集合 W 中的每一个元素 (w_j, t_j) 执行以下操作：
   - 计算其哈希值 $H(w_j)$。
   - 用自己的私钥 k_2 对哈希值进行指数运算，得到 $H(w_j)^{k_2}$。
   - 使用 P2 自己生成的同态加密公钥 pk 对关联值 t_j 进行加密，得到 $\mathrm{AEnc}(t_j)$。
4. P2 将所有处理后的数据对 $\{(H(w_j)^{k_2}, \mathrm{AEnc}(t_j))\}_{j=1}^{m_2}$ 以随机顺序发送给 P1。

核心思想: 只有当 $v_i = w_j$ 时，才会有 $H(v_i)^{k_1 k_2} = (H(w_j)^{k_2})^{k_1}$。P1 将在下一轮中利用此特性来识别交集。

5. 轮次 3 (Round 3)

此轮的目的是 P1 识别交集，并利用同态加密性质计算最终的和。

1. 对于从 P2 处收到的每一个数据对 $(H(w_j)^{k_2}, \mathrm{AEnc}(t_j))$，P1 用自己的私钥 k_1 对第一部分进行指数运算，得到 $(H(w_j)^{k_2})^{k_1} = H(w_j)^{k_1 k_2}$。
2. P1 现在可以识别出交集。P1 将在第 1 步中计算出的值与在 Round 2 中从 P2 收到的集合 Z 进行比较。如果某个 $H(w_j)^{k_1 k_2}$ 出现在集合 Z 中，则说明 w_j 必然是某个 v_i（因为哈希碰撞的概率极低），因此 w_j 属于交集。P1 由此构建一个索引集 J：

$$J = \{j : H(w_j)^{k_1 k_2} \in Z\}$$

其中 $Z = \{H(v_i)^{k_1 k_2}\}_{i=1}^{m_1}$。

3. P1 找出所有属于交集的项（即索引在 J 中的项），并利用加法同态加密的性质将它们对应的密文相加。

$$\mathrm{AEnc}(pk, S_J) = \mathrm{ASum}(\{\mathrm{AEnc}(t_j)\}_{j\in J}) = \mathrm{AEnc}\left(\sum_{j\in J} t_j\right)$$

这里的 S_J 就是交集元素的对应值之和 $\sum_{j\in J} t_j$。ASum 是同态加密方案的同态加法操作。

4. P1 使用 'ARefresh' 算法对求和后的密文进行随机化处理（rerandomizes），然后将最终的密文 $\mathrm{AEnc}(pk, S_J)$ 发送给 P2。随机化是为了防止 P2 从最终密文中推断出任何关于中间和的信息。

6. 输出 (Output)

- P2 接收到来自 P1 的最终加密总和。
- P2 使用自己的同态加密私钥 sk 对其进行解密，从而恢复出交集总和 S_J。

$$S_J = \mathrm{ADec}(sk, \mathrm{AEnc}(pk, S_J))$$

2.2 隐私性分析

- **P1 的隐私**: P2 只能看到被 k_1 盲化过的数据 $H(v_i)^{k_1}$。在 DDH 假设下，从 $H(v_i)^{k_1}$ 无法恢复出 $H(v_i)$，因此也无法知道 v_i。P2 同样无法知道哪些元素属于交集。
- **P2 的隐私**: P1 只能看到被 k_2 盲化过的数据 $H(w_j)^{k_2}$ 和被同态公钥加密过的数据 $\mathrm{AEnc}(t_j)$。在 DDH 假设下，P1 无法恢复 $H(w_j)$。由于同态加密是语义安全的，P1 无法解密 $\mathrm{AEnc}(t_j)$ 来获取任何关于 t_j 的信息。P1 只能知道哪些 w_j 在交集中，但不知道具体的 w_j 值，也不知道非交集元素的任何信息。最终 P1 只计算了加密后的总和，对具体值一无所知。

3 实验流程

代码通过面向对象的方式，将协议的两个参与方 P1 和 P2 分别封装在 `Party1` 和 `Party2` 两个类中。这种设计清晰地隔离了各自的私有数据（如密钥和原始数据集）和行为（协议中各轮次的操作）。

3.1 密码学原语的实现

- **循环群** G: 代码通过选取一个大的素数 `p`，并在此素数域上的乘法群 $\mathbb{Z}_p^*$ 中进行所有模指数运算（`pow(base, exp, p)`），来模拟协议中定义的素数阶循环群 G。这是一个为了功能验证而采取的简化实现。

- **哈希函数** H: 使用了标准的 `hashlib.sha256` 函数。为了将任意字符串的哈希结果映射到群 G 中，代码将 SHA256 的十六进制输出转换为一个大整数，然后对素数 `p` 取模。
- **加法同态加密 (AEnc)**: 直接使用了 `phe` 库中的 Paillier 加密方案。
  - `paillier.generate_paillier_keypair()` 用于生成密钥对 (pk, sk)。
  - `pk.encrypt(m)` 用于加密。
  - 密文之间的 `+` 运算符被库重载,用于执行同态加法,即 $\text{AEnc}(m_1)+\text{AEnc}(m_2) = \text{AEnc}(m_1 + m_2)$。
  - `sk.decrypt(c)` 用于解密。

3.2 核心协议逻辑的映射

- **Setup**: `Party2` 的 `setup` 方法生成了私钥 k_2 和同态密钥对 (pk, sk)，并将 pk 返回，模拟了发送给 P1 的过程。P1 在自己的 `setup` 方法中生成私钥 k_1。
- **Round 1**: 在 `Party1.round1()` 中,P1 对自己的每个元素 v_i 计算 $H(v_i)^{k_1} \pmod p$，实现了第一次盲化。函数返回前通过 `random.shuffle` 打乱列表，确保了协议的隐私性要求。
- **Round 2**: 在 `Party2.round2()` 中，P2 执行了双重任务。
  1. 它接收 P1 的数据,并用自己的私钥 k_2 进行二次盲化,计算出 $Z = \{(H(v_i)^{k_1})^{k_2} \pmod p\}$。
  2. 它处理自己的数据集 W,对每一个 (w_j, t_j),计算出盲化标识 $H(w_j)^{k_2} \pmod p$ 和加密值 $\text{AEnc}(t_j)$。

  这两组数据同样在返回前被打乱顺序。
- **Round 3**: 这是协议中最关键的一步，在 `Party1.round3()` 中实现。
  1. P1 对收到的每个 $H(w_j)^{k_2}$ 用自己的私钥 k_1 进行再盲化，得到 $(H(w_j)^{k_2})^{k_1} \pmod p$。
  2. **交集识别**: 通过比较 P1 自己计算的 $H(w_j)^{k_1 k_2}$ 是否存在于从 P2 处收到的集合 Z 中，来识别出交集元素。这是整个协议的核心机制，利用了 $H(v_i)^{k_1 k_2} = H(w_j)^{k_1 k_2} \iff v_i = w_j$。
  3. **同态求和**: P1 收集所有交集项对应的密文 $\text{AEnc}(t_j)$，并利用 `phe` 库提供的 `+` 运算，将它们累加成一个单一的密文 $\text{AEnc}(\sum t_j)$。
  4. **随机化 (ARefresh)**: 在将最终密文发回给 P2 之前，通过与一个加密的“0”相加，对结果进行了再随机化，这增强了安全性。

- **Output**: P2 在 `decrypt_final_sum()` 方法中使用自己的私钥 sk 解密 P1 发来的最终密文，从而得到明文总和 S_J。

4 实验结果

4.1 实验设置

在 `main` 函数中，我们定义了以下初始数据集：

- P1 的集合 V: `{'apple', 'banana', 'orange', 'grape', 'mango'}`
- P2 的数据字典 W: `{'banana': 10, 'grape': 25, 'pear': 15, 'apple': 5, 'watermelon': 30}`

基于以上数据，我们可以预先计算出理论上的正确结果：

- **理论交集**: P1 和 P2 集合的交集是 `{'apple', 'banana', 'grape'}`。
- **期望总和**: 交集元素在 P2 处对应的值的总和为 $10(\text{banana})+25(\text{grape})+5(\text{apple}) = 40$。

4.2 运行输出摘要

代码运行后，控制台输出模拟了双方的交互，并报告了最终结果如下：

```
PS D:\2. SDU\01.课程资料\6.大三下\实践\6\script> & S:/MACRO/anaconda/python.exe "d:/2. SDU/01.课程资料/6.大三下/实践/6/script/ddh.py"
==================================================
协议开始
==================================================
P1 的集合 V: {'banana', 'mango', 'apple', 'grape', 'orange'}
P2 的集合 W: {'banana': 10, 'grape': 25, 'pear': 15, 'apple': 5, 'watermelon': 30}
理论交集: {'banana', 'grape', 'apple'}
期望的总和: 40
--------------------------------------------------
公共参数设置完成。
P1 已初始化，持有 5 个元素。
P2 已初始化，持有 5 个元素。
P1: 已生成私钥 k1。
P2: 已生成私钥 k2 和同态加密密钥对 (pk, sk)。
P1: 已接收到同态加密公钥 pk。

--- P1: 开始 Round 1 ---
P1: 计算了 5 个盲化元素，并将其随机排序后发送给 P2。

--- P2: 开始 Round 2 ---
P2: 计算了集合 Z 并随机排序。
P2: 处理了自己的 5 个数据对，并将其随机排序。
P2: 将 Z 和处理后的数据对发送给 P1。

--- P1: 开始 Round 3 ---
P1: 对 P2 发来的 5 个数据对的第一部分进行再盲化。
P1: 发现 3 个交集元素。
P1: 已同态计算出交集值的总和。
P1: 对加密总和进行随机化，并发送给 P2。

--- P2: 输出阶段 ---
P2: 已解密最终结果。

==================================================
协议结束
==================================================
协议计算出的交集总和: 40
期望的交集总和: 40

[SUCCESS] 协议成功执行，结果正确！
```

图 1: 协议执行结果

4.3 结果分析

- **正确性**: 实验的最终输出“协议计算出的交集总和: 40”与我们预先计算的“期望的总和: 40”完全一致。这有力地证明了代码实现准确无误地执行了协议的逻辑。P1 成功地在不知道具体值的情况下，找到了交集并正确地聚合了对应的密文；P2 也成功地解密得到了最终的和。
- **隐私保护的体现**: 在整个交互过程中，P1 看到的 P2 数据是 $H(w_j)^{k_2}$ 和 $\mathrm{AEnc}(t_j)$，无法反解出 w_j 和 t_j。同样，P2 看到的 P1 数据是 $H(v_i)^{k_1}$，也无法反解出 v_i。双方的原始数据集除了交集的存在性（P1 可知，P2 不可知）和最终的和（P2 可知，P1 不可知）之外，没有泄露其他任何信息，符合协议的隐私保护目标。

附录 A 协议代码

```python
import random
import hashlib
from phe import paillier

# —— 1. 公共参数和辅助函数 ——

def setup_group_and_hash(key_length=1024):
    """
    生成一个大素数来模拟素数阶循环群 G。
    在实践中，会使用更复杂的密码学群。
    这里我们使用模 p 的乘法群。
    """
    # 为了简化，我们只使用一个大素数 p 作为模数。
    # 在一个真实的系统中，会使用安全素数和生成元。
    p = 
        100663435183797483443429381292534639899738483252199994361993313939369945300139

    # 定义一个哈希函数 H: U -> G
    # 将字符串输入哈希，然后转换为一个整数，最后模 p
    def H(identifier: str) -> int:
        hex_hash = hashlib.sha256(identifier.encode()).hexdigest()
        int_hash = int(hex_hash, 16)
        return int_hash % p

    print("公共参数设置完成。")
```

```
    return p, H

# —— 2. 参与方类的定义 ——

class Party1:
    def __init__(self, V: set, group_p: int, hash_function):
        """
        初始化 P1。
        :param V: P1 的标识符集合。
        :param group_p: 群的模数 p。
        :param hash_function: 哈希函数 H。
        """
        self.V = V
        self.p = group_p
        self.H = hash_function
        self.k1 = None
        self.pk = None
        print(f"P1 已初始化，持有 {len(self.V)} 个元素。")

    def setup(self):
        """P1 选择它的私钥 k1。"""
        # 从 Z_p^* 中选择一个随机私钥 k1
        self.k1 = random.randint(1, self.p - 1)
        print("P1: 已生成私钥 k1。")

    def receive_pk(self, pk: paillier.PaillierPublicKey):
        """P1 接收来自 P2 的同态加密公钥。"""
        self.pk = pk
        print("P1: 已接收到同态加密公钥 pk。")

    def round1(self) -> list:
        """
        执行协议的 Round 1。
        P1 对其集合中的每个元素 v 进行 H(v)^k1 计算。
        """
        print("\n—— P1: 开始 Round 1 ——")
        blinded_elements = [pow(self.H(v), self.k1, self.p) for v in
            self.V]
        random.shuffle(blinded_elements)
```

```
62          print(f"P1: 计算了 {len(blinded_elements)} 个盲化元素，并将其随机排序后发送给 P2。")
63          return blinded_elements
64
65      def round3(self, Z: list, p2_data: list) -> paillier.EncryptedNumber:
66          """
67          执行协议的 Round 3。
68          P1 识别交集并计算加密后的总和。
69          :param Z: 从 P2 收到的 H(v_i)^(k1*k2) 集合。
70          :param p2_data: 从 P2 收到的 (H(w_j)^k2, AEnc(t_j)) 对的列表。
71          :return: 加密后的最终总和 S_J。
72          """
73          print("\n—— P1: 开始 Round 3 ——")
74
75          # 1. P1 对收到的 p2_data 中的第一部分进行指数运算
76          p1_reblinded_map = {
77              pow(h_w_k2, self.k1, self.p): encrypted_t
78              for h_w_k2, encrypted_t in p2_data
79          }
80          print(f"P1: 对 P2 发来的 {len(p2_data)} 个数据对的第一部分进行再盲化。")
81
82          # 2. 识别交集
83          # 将 Z 转换为集合以便快速查找
84          Z_set = set(Z)
85          intersection_ciphertexts = []
86          for reblinded_h, encrypted_t in p1_reblinded_map.items():
87              if reblinded_h in Z_set:
88                  intersection_ciphertexts.append(encrypted_t)
89
90          print(f"P1: 发现 {len(intersection_ciphertexts)} 个交集元素。")
91
92          # 3. 同态地计算总和
93          if not intersection_ciphertexts:
94              # 如果没有交集，返回加密的0
95              encrypted_sum = self.pk.encrypt(0)
```

```
96          else:
97              # 使用同态加法将所有交集元素的密文相加
98              encrypted_sum = intersection_ciphertexts[0]
99              for i in range(1, len(intersection_ciphertexts)):
100                 encrypted_sum += intersection_ciphertexts[i]
101
102         print("P1: 已同态计算出交集值的总和。")
103
104         # 4. 随机化结果 (ARefresh)
105         # Paillier 的同态加法已经具有随机性，但为了严格遵循协议，
106         # 我们可以通过加上一个加密的 0 来显式地进行再随机化。
107         refreshed_encrypted_sum = encrypted_sum + self.pk.encrypt(0)
108         print("P1: 对加密总和进行随机化，并发送给 P2。")
109
110         return refreshed_encrypted_sum
111
112
113 class Party2:
114     def __init__(self, W: dict, group_p: int, hash_function,
        he_key_length=1024):
115         """
116         初始化 P2。
117         :param W: P2 的 {标识符: 值} 字典。
118         :param group_p: 群的模数 p。
119         :param hash_function: 哈希函数 H。
120         """
121         self.W = W
122         self.p = group_p
123         self.H = hash_function
124         self.k2 = None
125         self.pk = None
126         self.sk = None
127         self.he_key_length = he_key_length
128         print(f"P2 已初始化，持有 {len(self.W)} 个元素。")
129
130     def setup(self):
131         """P2 选择私钥 k2 并生成同态加密密钥对。"""
132         # 选择私钥 k2
133         self.k2 = random.randint(1, self.p - 1)
```

```
134         # 生成同态加密密钥对
135         self.pk, self.sk = paillier.generate_paillier_keypair(
                n_length=self.he_key_length)
136         print("P2: 已生成私钥 k2 和同态加密密钥对 (pk, sk)。")
137         return self.pk
138
139     def round2(self, p1_data: list) -> (list, list):
140         """
141         执行协议的 Round 2。
142         :param p1_data: 从 P1 处收到的 H(v_i)^k1 列表。
143         :return: (Z, p2_prepared_data) 元组
144         """
145         print("\n—— P2: 开始 Round 2 ——")
146
147         # 1. 计算 Z = {H(v_i)^(k1*k2)}
148         Z = [pow(h_v_k1, self.k2, self.p) for h_v_k1 in p1_data]
149         random.shuffle(Z)
150         print(f"P2: 计算了集合 Z 并随机排序。")
151
152         # 2. 处理自己的数据 W
153         p2_prepared_data = []
154         for w, t in self.W.items():
155             # 计算 H(w_j)^k2
156             h_w_k2 = pow(self.H(w), self.k2, self.p)
157             # 加密 t_j
158             encrypted_t = self.pk.encrypt(t)
159             p2_prepared_data.append((h_w_k2, encrypted_t))
160
161         random.shuffle(p2_prepared_data)
162         print(f"P2: 处理了自己的 {len(self.W)} 个数据对，并将其随机排
                序。")
163
164         print("P2: 将 Z 和处理后的数据对发送给 P1。")
165         return Z, p2_prepared_data
166
167     def decrypt_final_sum(self, final_ciphertext: paillier.
            EncryptedNumber) -> int:
168         """
169         解密从 P1 处收到的最终密文。
```

```
        """
        print("\n—— P2: 输出阶段 ——")
        decrypted_sum = self.sk.decrypt(final_ciphertext)
        print(f"P2: 已解密最终结果。")
        return decrypted_sum

# —— 3. 协议执行流程 ——

def main():
    # 定义双方的数据
    # V: P1 的集合
    # W: P2 的 {标识符: 值} 字典
    p1_items = {'apple', 'banana', 'orange', 'grape', 'mango'}
    p2_items = {'banana': 10, 'grape': 25, 'pear': 15, 'apple': 5, '
        watermelon': 30}

    # 计算期望结果
    intersection_keys = p1_items.intersection(p2_items.keys())
    expected_sum = sum(p2_items[key] for key in intersection_keys)

    print("="*50)
    print("协议开始")
    print("="*50)
    print(f"P1 的集合 V: {p1_items}")
    print(f"P2 的集合 W: {p2_items}")
    print(f"理论交集: {intersection_keys}")
    print(f"期望的总和: {expected_sum}")
    print("-" * 50)

    # 1. Setup
    p, H = setup_group_and_hash()

    p1 = Party1(p1_items, p, H)
    p2 = Party2(p2_items, p, H)

    p1.setup()
    p2_pk = p2.setup()
    p1.receive_pk(p2_pk)

```

```
    # 2. Round 1
    p1_to_p2_data = p1.round1()

    # 3. Round 2
    Z, p2_to_p1_data = p2.round2(p1_to_p2_data)

    # 4. Round 3
    final_encrypted_sum = p1.round3(Z, p2_to_p1_data)

    # 5. Output
    final_sum = p2.decrypt_final_sum(final_encrypted_sum)

    print("\n" + "="*50)
    print("协议结束")
    print("="*50)
    print(f"协议计算出的交集总和: {final_sum}")
    print(f"期望的交集总和: {expected_sum}")

    # 验证结果
    if final_sum == expected_sum:
        print("\n[SUCCESS] 协议成功执行, 结果正确! ")
    else:
        print("\n[FAILURE] 协议执行失败, 结果不匹配。")

if __name__ == "__main__":
    main()
```