



网络空间安全创新创业实践 Project6

吴扬

2025 年 7 月 31 日

目录

1 实验环境	2
2 实验原理	2
2.1 核心思想：OPRF	2
2.2 协议的数学流程	2
2.3 隐私集合交集 (PSI) 的应用	3
3 实验步骤	3
3.1 步骤一：全局配置与辅助函数	4
3.2 步骤二：服务器端 (Server) 实现	6
3.3 步骤三：客户端 (Client) 实现	8
4 实验结果呈现	10
5 心得感悟	12

1 实验环境

处理器	Intel(R) Core(TM) i9-14900HX 2.20 GHz
机带 RAM	16.0 GB (15.6 GB 可用)
Windows 版本	Windows 11 家庭中文版
操作系统版本	22631.4169

2 实验原理

现代网络环境中，密码泄露事件频发。用户迫切需要一种机制来检查自己的密码是否已泄露，但直接将密码发送给服务商（如 Google）进行查询会带来巨大的隐私风险。Google Password Checkup 通过一个精妙的密码学协议解决了这个难题，其核心是基于椭圆曲线盲签名的 OPR。

2.1 核心思想：OPRF

OPRF 允许两方（客户端和服务器）联合计算一个伪随机函数 $F_k(x)$ 的值，其中服务器持有密钥 k ，客户端持有输入 x 。协议结束后，客户端得到正确的输出 $F_k(x)$ ，但服务器对客户端的输入 x 一无所知；同时，客户端对服务器的密钥 k 也一无所知。这正是“茫然”的含义。

2.2 协议的数学流程

协议依赖于椭圆曲线密码学 (ECC) 中的标量乘法（点乘）运算。设 P 是椭圆曲线上的一点， s 是一个整数（标量），它们的标量乘法记为 $s \cdot P$ ，结果仍是曲线上的一点。该运算满足交换律： $k \cdot (t \cdot P) = t \cdot (k \cdot P) = (k \cdot t) \cdot P$ 。

协议流程如下：

1. **客户端准备输入**: 客户端首先将自己的用户名 u 和密码 p 通过一个公开的哈希函数 H_1 转换成一个统一的哈希值 x 。

$$x = H_1(u, p)$$

接着，使用另一个哈希函数 H_2 （称为 Hash-to-Curve）将 x 映射到选定的椭圆曲线上的一点 P 。

$$P = H_2(x)$$

2. **客户端盲化 (Blind)**: 客户端生成一个随机的大整数，称为盲化因子 t 。然后，客户端计算“盲化点” T 并将其发送给服务器。

$$T = t \cdot P$$

由于 t 是随机且保密的，服务器无法从 T 中反推出原始的点 P 。

3. **服务器处理 (Evaluate)**: 服务器持有一个长期保密的私钥 k (也是一个大整数)。收到客户端发来的 T 后，服务器用自己的密钥 k 对其进行标量乘法运算，得到结果点 Z ，并将其返回给客户端。

$$Z = k \cdot T = k \cdot (t \cdot P)$$

4. **客户端去盲 (Unblind)**: 客户端收到服务器返回的 Z 后，需要消除自己之前加入的盲化因子 t 的影响。这通过乘以 t 的模逆元 $t^{-1} \pmod{n}$ 来实现 (n 是椭圆曲线基点的阶)。

$$V = t^{-1} \cdot Z$$

根据椭圆曲线点乘的交换律，我们有：

$$V = t^{-1} \cdot (k \cdot t \cdot P) = (t^{-1} \cdot t) \cdot (k \cdot P) = 1 \cdot (k \cdot P) = k \cdot P$$

最终，客户端独立计算出了值 $V = k \cdot P = k \cdot H_2(H_1(u, p))$ 。这个值就是 OPRF 的输出 $F_k(x)$ 。整个过程中，服务器只看到了盲化点 T ，不知道 x ；客户端只知道自己的 x 和 t ，不知道服务器的 k 。

2.3 隐私集合交集 (PSI) 的应用

得到 OPRF 的输出后，客户端需要用它来和服务器的泄露数据库进行比对。

- 服务器对其数据库中所有已泄露的凭据 y_i 进行预处理，计算出它们对应的 OPRF 值： $\{V_i = k \cdot H_2(y_i)\}$ 。
- 为了节省带宽和提高效率，服务器并不直接提供完整的 V_i 列表。它将每个 V_i 转换为字节串后，取其一个较短的前缀（例如 4 个字节），并将所有不重复的前缀组成的列表公开给客户端。
- 客户端在本地计算出自己凭据的 OPRF 值 V 后，同样提取其前缀，并检查该前缀是否存在于从服务器获取的前缀列表中。
- 如果前缀匹配，说明该密码可能已泄露。为了消除因前缀碰撞导致的误报（假阳性），客户端会向服务器请求该前缀对应的所有完整哈希值，在本地进行一次最终的、完整的匹配确认。

3 实验步骤

本实验基于 Python 3，并选用 `tinyec` 库来实现椭圆曲线运算。代码结构主要分为两个类：`Server` 和 `Client`，以及一些全局辅助函数。

3.1 步骤一：全局配置与辅助函数

首先，定义协议所需的全局参数和工具函数。

- **曲线选择**: 使用 `tinyec.registry.get_curve('secp256r1')` 来获取标准化的 NIST P-256 椭圆曲线。客户端和服务器必须使用同一曲线。
- **哈希函数 H1**: 定义函数 `h1(username, password)`，它使用 `hashlib.sha256` 将用户名和密码的组合安全地哈希成一个字节串，作为协议的输入 `x`。
- **哈希到曲线函数 H2**: 定义函数 `h2_hash_to_curve(data)`，这是实现中的一个关键。它负责将任意字节串 `data` 映射到椭圆曲线上的一个有效点。本实验采用“尝试-递增”(Try and Increment) 的简单方法：将输入哈希为一个候选的 `x` 坐标，然后根据曲线方程 $y^2 = x^3 + ax + b \pmod{p}$ 尝试计算出 `y` 坐标。如果成功，则返回该点；如果失败，则在输入后附加一个空字节再次哈希，直到找到有效点。
- **序列化函数**: 定义 `point_to_bytes` 和 `bytes_to_point` 函数。由于 `tinyec` 的点对象不能直接在网络上传输，我们需要定义一个统一的格式（例如，将点的 `x, y` 坐标转为字节串并拼接）来模拟数据的发送和接收。

```
1 # 1. 定义使用的椭圆曲线
2 # 我们从 tinyec 的注册表中获取标准曲线 'secp256r1' (NIST P-256)
3 CURVE = registry.get_curve('secp256r1')
4
5 # 2. 定义哈希函数 H1 (此函数不变)
6 def h1(username: str, password: str) -> bytes:
7     """
8         哈希函数 H1: 将用户名和密码组合并哈希，作为协议的输入 x。
9     """
10    username_bytes = username.encode('utf-8')
11    password_bytes = password.encode('utf-8')
12    hasher = hashlib.sha256()
13    hasher.update(b"username:" + username_bytes)
14    hasher.update(b"password:" + password_bytes)
15    return hasher.digest()
16
17 # 3. 定义哈希函数 H2 (Hash-to-Curve)
18 # H2 的实现针对 tinyec 库进行了修改。
19 def h2_hash_to_curve(data: bytes) -> Point:
20     """
21         哈希函数 H2: 将字节串 data 映射到椭圆曲线 CURVE 上的一个点。
22     """
```

```
22 现在返回一个 tinyec 的 Point 对象。
23 """
24 # tinyec 提供了直接从哈希值生成点的方法，更简单健壮
25 # 我们只需要确保哈希的输出（整数）在曲线的域内
26 while True:
27     # 通过哈希 data 生成候选的 x 坐标
28     x_bytes = hashlib.sha256(data).digest()
29     x = int.from_bytes(x_bytes, 'big')
30
31     # 尝试在曲线上找到与 x 对应的点
32     # 曲线方程:  $y^2 = x^3 + ax + b \pmod{p}$ 
33     # 我们需要计算右边  $y_{\text{squared}} = (x^3 + a*x + b) \pmod{p}$ 
34     y_sq = (pow(x, 3, CURVE.field.p) + CURVE.a * x + CURVE.b) %
35             CURVE.field.p
36
37     # 计算模 p 的平方根来找到 y
38     # 注意：不是所有数都有模平方根。这里使用 Tonelli-Shanks 或
39     # Cipolla's 算法
40     # 为了简化，tinyec 内部可能处理了，但一个简单的方法是检查它
41     # 是否是二次剩余
42     if pow(y_sq, (CURVE.field.p - 1) // 2, CURVE.field.p) == 1:
43         # 如果是二次剩余，我们可以计算出 y
44         y = pow(y_sq, (CURVE.field.p + 1) // 4, CURVE.field.p) #
45             适用于  $p \equiv 3 \pmod{4}$ 
46
47         # 创建并返回点对象
48         return Point(CURVE, x, y)
49
50     # 如果找不到点，就附加一个字节再次哈希 (Try and Increment)
51     data += b'\x00'
52
53 # 4. 序列化和反序列化函数
54 # 我们需要一种方法将 tinyec 的点对象转换为字节串以便网络传输，反之亦然。
55 def point_to_bytes(point: Point) -> bytes:
56     """将 tinyec Point 对象序列化为字节串。"""
57     # 我们简单地将 x 和 y 坐标拼接起来
58     coord_size = (CURVE.field.n.bit_length() + 7) // 8
59     return point.x.to_bytes(coord_size, 'big') + point.y.to_bytes(
```

```

    coord_size, 'big')

56
57 def bytes_to_point(b: bytes) -> Point:
58     """从字节串反序列化为 tinyec Point 对象。"""
59     coord_size = (CURVE.field.n.bit_length() + 7) // 8
60     x = int.from_bytes(b[:coord_size], 'big')
61     y = int.from_bytes(b[coord_size:], 'big')
62     return Point(CURVE, x, y)
63
64 # 5. 定义前缀长度（此部分不变）
65 PREFIX_LENGTH_BYTES = 4

```

3.2 步骤二：服务器端 (Server) 实现

Server 类模拟了服务提供商的后台。

- **初始化 (`__init__`):**
 1. 生成一个服务器私有的、长期持有的秘密密钥 `_k`。它是一个在曲线的阶范围内的随机整数。
 2. 对外来的“泄露凭据数据库”`breached_credentials` 进行预处理。遍历数据库中的每一条记录 y_i ，计算其 OPRF 值 $V_i = _k \cdot H_2(y_i)$ 。
 3. 将计算出的 V_i 按其前缀进行分组存储在一个字典 `_breached_prf_values` 中，键为前缀，值为包含该前缀的所有完整哈希列表。
- **处理盲化请求 (`handle_blinded_request`):** 这是服务器的核心交互接口。它接收客户端发来的盲化点（的字节串形式），将其反序列化为点对象 T ，然后执行点乘运算 $Z = _k \cdot T$ ，最后将结果 Z 序列化后返回给客户端。
- **提供数据接口:**
 - `get_breached_prf_prefixes()`: 返回预处理好的所有泄露数据的前缀列表。
 - `get_full_hashes_for_prefix(prefix)`: 当客户端报告前缀匹配时，返回该前缀对应的所有完整哈希值，用于最终确认。

```

1 class Server:
2
3     def __init__(self, breached_credentials: Set[Tuple[str, str]]):
4         print("--- [服务器] 初始化开始 ---")
5         # 1. 生成服务器的秘密密钥 k
6         # 在 tinyec 中，密钥 k 就是一个在曲线阶范围内的随机整数

```

```
6     self._k = int.from_bytes(os.urandom(32), 'big') % CURVE.  
7         field.n  
8     print(f"[服务器] 秘密密钥 k 已生成。")  
9  
10    self._breached_prf_values = {}  
11    print(f"[服务器] 正在预处理 {len(breached_credentials)} 条泄  
12        露凭据...")  
13  
14    for username, password in breached_credentials:  
15        y = h1(username, password)  
16        h2_y = h2_hash_to_curve(y)  
17  
18        # 计算 PRF 值: v_y = k * H2(y)  
19        # 使用 tinyec, 点乘法可以直接用 * 操作符, 非常直观!  
20        v_y_point = self._k * h2_y  
21  
22  
23        v_y_bytes = point_to_bytes(v_y_point)  
24        prefix = v_y_bytes[:PREFIX_LENGTH_BYTES]  
25  
26  
27        if prefix not in self._breached_prf_values:  
28            self._breached_prf_values[prefix] = []  
29        self._breached_prf_values[prefix].append(v_y_bytes)  
30  
31  
32        print(f"[服务器] 预处理完成。生成了 {len(self.  
33            _breached_prf_values)} 个唯一的前缀。")  
34        print("--- [服务器] 初始化结束 ---\n")  
35  
36  
37    def get_breached_prf_prefixes(self) -> List[bytes]:  
38        print("[服务器] 收到客户端请求, 发送所有泄露凭据的PRF值前缀  
39            列表。")  
40        return list(self._breached_prf_values.keys())  
41  
42  
43    def handle_blinded_request(self, T_bytes: bytes) -> bytes:  
44        print("[服务器] 收到客户端的盲化请求 T。")  
45        # 1. 从字节串恢复出点 T  
46        T_point = bytes_to_point(T_bytes)  
47  
48        # 2. 计算 Z = k * T, 这里的乘法现在可以正确工作了  
49        Z_point = self._k * T_point
```

```

41
42     # 3. 将结果点 Z 转换回字节串
43     Z_bytes = point_to_bytes(Z_point)
44     print("[服务器] 计算完成 Z = k * T, 并将其返回给客户端。")
45     return Z_bytes
46
47     def get_full_hashes_for_prefix(self, prefix: bytes) -> List[
48         bytes]:
49         print(f"[服务器] 收到客户端对前缀 {prefix.hex()} 的请求, 返回
          对应的完整哈希。")
50         return self._breached_prf_values.get(prefix, [])

```

3.3 步骤三：客户端 (Client) 实现

Client 类模拟了用户浏览器中的 Password Checkup 功能。

- **初始化 (`__init__`)**: 存储用户要检查的用户名和密码。
- **执行检查 (`check_password_leak`)**: 该方法完整地执行了协议的客户端逻辑。

1. 盲化阶段:

- 计算 $x = H_1(u, p)$ 和 $P = H_2(x)$ 。
- 生成一个一次性的随机整数作为盲化因子 t 。
- 计算盲化点 $T = t \cdot P$ 。

2. 交互阶段:

- 将 T 序列化后发送给服务器的 `handle_blinded_request` 接口。
- 接收服务器返回的结果 Z 。

3. 去盲与检查阶段:

- 计算 t 的模逆元 t^{-1} 。
- 计算最终的 OPRF 值 $V = t^{-1} \cdot Z$ 。
- 从服务器获取前缀列表，并检查 V 的前缀是否在其中。
- 若前缀匹配，则请求该前缀对应的所有完整哈希，进行本地比对，得出最终结论。若不匹配，则密码是安全的。

```

1 class Client:
2
3     def __init__(self, username: str, password: str):
4         self.username = username
5         self.password = password

```

```
5     print(f"--- [客户端] 初始化，准备检查凭据（用户: '{self.
6         username}'） ---\n")
7
8     def check_password_leak(self, server: Server) -> bool:
9         print(f"[客户端] 开始检查用户 '{self.username}' 的密码。")
10
11     # === 协议第一步：客户端进行盲化 ===
12     x = h1(self.username, self.password)
13     print(f"[客户端] 1. 计算凭据哈希 H1(u, p) = {x.hex()}")
14
15     P = h2_hash_to_curve(x)
16     print(f"[客户端] 2. 将哈希映射到曲线点 P")
17
18     # 盲化因子 t 是一个随机整数
19     t = int.from_bytes(os.urandom(32), 'big') % CURVE.field.n
20     print(f"[客户端] 3. 生成随机盲化因子 t")
21
22     # 计算盲化点 T = t * P，现在可以正确工作
23     T_point = t * P
24     T_bytes = point_to_bytes(T_point)
25     print(f"[客户端] 4. 计算盲化点 T = t * P，准备发送给服务器。
26           ")
27
28     # === 协议第二步：与服务器交互 ===
29     print("\n[客户端] --- 开始与服务器通信 ---")
30     Z_bytes = server.handle_blinded_request(T_bytes)
31     print("[客户端] --- 通信结束 ---\n")
32
33     # === 协议第三步：客户端去盲并检查 ===
34     # 计算 t 的模逆元，注意模数是曲线的阶 CURVE.field.n
35     t_inv = pow(t, -1, CURVE.field.n)
36     print(f"[客户端] 5. 计算 t 的逆元 t_inv")
37
38     Z_point = bytes_to_point(Z_bytes)
39
40     # 去盲，计算 V = t_inv * Z
41     V_point = t_inv * Z_point
42     V_bytes = point_to_bytes(V_point)
43     print(f"[客户端] 6. 去盲得到 PRF 值 V = k*H2(x) = {V_bytes}.
```

```
        hex()})"

42
43     leaked_prefixes = server.get_breached_prf_prefixes()
44     print(f"[客户端] 7. 从服务器获取了 {len(leaked_prefixes)} 个
45         泄露数据的前缀。")

46     my_prefix = V_bytes[:PREFIX_LENGTH_BYT
47     print(f"[客户端] 8. 我计算出的 V 的前缀是: {my_prefix.hex()}

48
49     if my_prefix in leaked_prefixes:
50         print(f"[客户端] 9. 警告! 前缀匹配成功。可能存在泄露, 需
51             要进一步确认。")
52
53         full_hashes_for_prefix = server.
54             get_full_hashes_for_prefix(my_prefix)
55
56         if V_bytes in full_hashes_for_prefix:
57             print(f"[客户端] 10. 最终确认: 完整的 PRF 值匹配成
58                 功。")
59             print(f"--- [结论] 凭据 (用户: '{self.username}') 已
60                 经泄露! ---\n")
61             return True
62
63         else:
64             print(f"[客户端] 10. 最终确认: 完整的 PRF 值不匹配。
65                 这是一次前缀碰撞, 凭据安全。")
66             print(f"--- [结论] 凭据 (用户: '{self.username}') 是
67                 安全的。 ---\n")
68             return False
69
70     else:
71         print(f"[客户端] 9. 前缀不匹配。")
72         print(f"--- [结论] 凭据 (用户: '{self.username}') 是安全
73             的。 ---\n")
74         return False
```

4 实验结果呈现

我们在主程序中设置了一个包含 4 条泄露记录的模拟数据库，并模拟了两种场景来测试协议的有效性。

- 场景 1: 客户端检查一个确定在泄露数据库中的凭据 ('alice', '123456')。

- **场景 2:** 客户端检查一个安全的、不在泄露数据库中的凭据 ('eve', 'MySecurePa\$\$w0rd')。

运行 main.py 脚本后的输出结果截图如下所示。

```
sherlock@ubuntu:~/study/google$ python3 main.py
-- [服务器] 初始化开始 ---
[服务器] 秘密密钥 k 已生成。
[服务器] 正在预处理 4 条泄露凭据...
[服务器] 预处理完成。生成了 4 个唯一的前缀。
-- [服务器] 初始化结束 ---

=====
场景1: 检查一个已泄露的密码 ('alice', '123456')
=====
-- [客户端] 初始化, 准备检查凭据 (用户: 'alice') ---

[客户端] 开始检查用户 'alice' 的密码。
[客户端] 1. 计算凭据哈希 H1(u, p) = 16ac5fbf1b5b9bc6f6226527f46fb54703ac66b991778481b0703b8c0bc8e12
[客户端] 2. 将哈希映射到曲线点 P
[客户端] 3. 生成随机盲化因子 t
[客户端] 4. 计算盲化点 T = t * P, 准备发送给服务器。

[客户端] --- 开始与服务器通信 ---
[服务器] 收到客户端的盲化请求 T。
[服务器] 计算完成 Z = k * T, 并将其返回给客户端。
[客户端] --- 通信结束 ---

[客户端] 5. 计算 t 的逆元 t_inv
[客户端] 6. 去盲得到 PRF 值 V = k*H2(x) = 3196694dba21914c2f122f9bcee8a45e62a639807660402d493d0388459b96cddf212d8f50b17732c5d5f7e2ef0b1fd6aad293350074f314c0692e48da1b241e
[服务器] 收到客户端请求, 发送所有泄露凭据的PRF值前缀列表。
[客户端] 7. 从服务器获取了 4 个泄露数据的前缀。
[客户端] 8. 我计算出的 V 的前缀是: 3196694d
[客户端] 9. 警告! 前缀匹配成功。可能存在泄露, 需要进一步确认。
[服务器] 收到客户端对前缀 3196694d 的请求, 返回对应的完整哈希。
[客户端] 10. 最终确认: 完整的 PRF 值匹配成功。
-- [结论] 凭据 (用户: 'alice') 已经泄露! --
```

图 1: 场景 1: 检查已泄露密码的运行结果

```
=====
场景2: 检查一个安全的密码 ('eve', 'MySecurePa$$w0rd')
=====

--- [客户端] 初始化, 准备检查凭据 (用户: 'eve') ---

[客户端] 开始检查用户 'eve' 的密码。
[客户端] 1. 计算凭据哈希 H1(u, p) = 75f53ce0e7386ecc764233e0be9259303a91963e4402da367529f9b4a8a2d
eff
[客户端] 2. 将哈希映射到曲线点 P
[客户端] 3. 生成随机盲化因子 t
[客户端] 4. 计算盲化点 T = t * P, 准备发送给服务器。

[客户端] --- 开始与服务器通信 ---
[服务器] 收到客户端的盲化请求 T。
[服务器] 计算完成 Z = k * T, 并将其返回给客户端。
[客户端] --- 通信结束 ---

[客户端] 5. 计算 t 的逆元 t_inv
[客户端] 6. 去盲得到 PRF 值 V = k*H2(x) = ff9a5c3b0143904f69e3331437fd7bf432872dab670e9fd086be91b
016c0f1b2e90991cf975a96d3e37084e0bb1cd3287629026833fb9410c006176376b5018
[服务器] 收到客户端请求, 发送所有泄露凭据的PRF值前缀列表。
[客户端] 7. 从服务器获取了 4 个泄露数据的前缀。
[客户端] 8. 我计算出的 V 的前缀是: ff9a5c3b
[客户端] 9. 前缀不匹配。
--- [结论] 凭据 (用户: 'eve') 是安全的。 ---
```

图 2: 场景 2: 检查安全密码的运行结果

实验结果清晰地表明：

- 对于场景 1 中的泄露凭据，客户端计算出的 PRF 值前缀与服务器列表中的一个前缀成功匹配，并通过后续的完整哈希比对，最终确认密码已泄露。
- 对于场景 2 中的安全凭据，客户端计算出的 PRF 值前缀未在服务器列表中找到，直接得出密码安全的结论。

整个过程完全符合协议设计，证明了我们实现的正确性。

5 心得感悟

本次实验让我深刻体会到了现代应用密码学的精妙与力量。它不仅仅是停留在理论上的数学公式，更是能够解决现实世界中复杂隐私问题的强大工具。

1. OPRF 协议的设计非常优雅。通过“盲化-处理-去盲”这一经典流程，它巧妙地在不经意间完成了两方协同计算，实现了看似不可能的隐私保护目标。这让我对密码学设计思想有了更深的理解，即如何利用数学工具的特性（如点乘交换律）来构建安全的交互流程。
2. 在早期的实现尝试中，我遇到了一个关键的挑战：密码学库的选择。最初尝试使用 pyca/cryptography 这个主流的库，但很快发现它的高层 API 为了安全，封装了底层的点运算，不允许开发者直接进行任意的点乘法。这个挫折让我明白，选择合适的工具至关重要。对于实现标准协议，高层封装的库更安全易用；而对于

构建和学习新协议，需要一个能提供底层原语操作的库，这也是最终选择 tinyec 的原因。这个过程本身就是一次宝贵的学习经历。

3. 协议中“使用前缀”的设计是一个典型的在安全性和效率之间做权衡的例子。发送完整哈希值最安全，但网络开销巨大；发送前缀大大减少了通信量，但引入了碰撞的可能性（假阳性）。协议通过“前缀匹配后，再请求完整哈希”的两步走策略，完美地解决了这个问题，在保证最终结果准确性的前提下，极大地优化了系统性能。这对于将密码学方案部署到像 Chrome 浏览器这样亿万用户级别的产品中是至关重要的。
4. 通过这个项目，我更加认识到隐私保护在当今数字社会的重要性。Google Password Checkup 是一个杰出的范例，它告诉我们，便捷的服务和强大的隐私保护可以并行不悖，而这背后的基石正是严谨而创新的密码学研究。