网络空间安全创新创业实践

Project4

姓　　名: 张治瑞
学　　号: 202200210078
班　　级: 网安 22.1 班
学　　院: 网络空间安全学院

2025 年 8 月 10 日

目录

1 实验环境

处理器	Intel(R) Core(TM) i9-14900HX 2.20 GHz
机载 RAM	16.0 GB (15.6 GB 可用)
Windows 版本	Windows 11

2 实验题目与目的

2.1 实验题目

Google Password Checkup: 基于盲化 PRF 的隐私保护密码泄露检测系统实现

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）

2.2 实验目的

1. 深入理解密码学中盲化技术的原理和应用
2. 掌握伪随机函数 (PRF) 在隐私保护中的作用机制
3. 实现完整的 Google Password Checkup 协议流程
4. 验证系统在保护用户隐私的同时准确检测密码泄露的能力
5. 分析协议的安全性、效率和实用性

3 理论基础与协议原理

3.1 问题背景

随着网络安全事件频发，大量用户凭据在数据泄露中被暴露。用户需要一种方法来检查自己的密码是否已经泄露，但同时不希望将明文密码发送给检查服务提供商。Google Password Checkup 协议解决了这个隐私保护与功能性之间的矛盾。

3.2 核心技术原理

3.2.1 盲化伪随机函数 (Blinded PRF)

设 $F_k : \{0,1\}^* \to \{0,1\}^n$ 为一个以 k 为密钥的伪随机函数。盲化 PRF 协议允许客户端在不向服务器泄露输入 x 的情况下，计算 $F_k(x)$。

核心思想是：

- 客户端使用随机数 t 对输入进行盲化：$T = t \cdot H_2(x)$
- 服务器对盲化输入计算 PRF：$Z = k \cdot T$
- 客户端去盲化获得结果：$V = t^{-1} \cdot Z = k \cdot H_2(x)$

3.2.2 椭圆曲线密码学基础

协议基于椭圆曲线 $y^2 = x^3 + ax + b \pmod{p}$ 实现：

- 使用 secp256r1 曲线，提供 128 位安全级别
- 利用椭圆曲线离散对数问题的困难性
- 支持高效的点运算和标量乘法

3.3 协议流程详述

预处理阶段：

1. 服务器生成私钥 k
2. 对每个泄露凭据 (u_i, p_i)：
    - 计算 $y_i = H_1(u_i, p_i)$
    - 计算 $P_i = H_2(y_i)$
    - 计算 $V_i = k \cdot P_i$
    - 存储 V_i 的前缀用于快速匹配

在线检测阶段：

1. **客户端盲化：**
    - 计算 $x = H_1(u, p)$
    - 计算 $P = H_2(x)$
    - 选择随机盲化因子 t
    - 计算盲化点 $T = t \cdot P$
    - 发送 T 给服务器
2. **服务器计算：**
    - 计算 $Z = k \cdot T$

- 发送 Z 给客户端

3. **客户端验证：**
   - 计算 $V = t^{-1} \cdot Z = k \cdot P$
   - 检查 V 的前缀是否在泄露列表中
   - 如匹配，进行完整哈希验证

4 系统设计与代码实现

4.1 系统架构设计

系统采用模块化设计，主要组件包括：

- **配置管理模块**：统一管理系统参数和常量
- **密码学操作模块**：封装核心密码学算法
- **服务器模块**：处理泄露数据库和 PRF 计算
- **客户端模块**：执行盲化和验证流程
- **异常处理模块**：提供完善的错误处理机制

4.2 关键代码模块分析

4.2.1 配置管理模块

Listing 1:

```python
@dataclass(frozen=True)
class Config:
    """系统配置类"""
    CURVE_NAME: str = 'secp256r1'           # 椭圆曲线类型
    PREFIX_LENGTH_BYTES: int = 4            # 前缀长度
    HASH_ALGORITHM: str = 'sha256'          # 哈希算法
    MAX_HASH_TO_CURVE_ATTEMPTS: int = 1000  # 最大重试次数
    DEFAULT_RANDOM_BYTES: int = 32          # 随机数长度
```

设计分析：

- 使用 `@dataclass(frozen=True)` 确保配置不可变性

- 选择 secp256r1 曲线平衡安全性和性能
- 4 字节前缀长度在安全性和效率间取得平衡
- 合理的重试次数避免 Hash-to-Curve 失败

4.2.2 哈希函数实现

Listing 2:

```python
def h1(self, username:  str, password:  str) -> bytes:
    """哈希函数 H1: 将用户名和密码组合并哈希"""
    try:
        validate_input(username, password)

        username_bytes = username.encode('utf-8')
        password_bytes = password.encode('utf-8')

        hasher = hashlib.new(self.config.HASH_ALGORITHM)
        hasher.update(b"username:" + username_bytes)
        hasher.update(b"password:" + password_bytes)

        return hasher.digest()
    except Exception as e:
        raise CryptographicError(f"H1哈希计算失败: {e}")
```

实现特点：

- 使用明确的前缀区分用户名和密码字段
- 严格的输入验证防止无效数据
- UTF-8 编码确保字符处理的一致性
- 完善的异常处理机制

4.2.3 Hash-to-Curve 核心算法

Listing 3:

```python
def h2_hash_to_curve(self, data: bytes) -> Point:
    """哈希函数 H2: 将字节串映射到椭圆曲线上的点"""
    attempts = 0
```

```
    working_data = data

    while attempts < self.config.MAX_HASH_TO_CURVE_ATTEMPTS:
        try:
            # 生成候选x坐标
            x_bytes = hashlib.new(self.config.HASH_ALGORITHM, working_data)
                .digest()
            x =  int.from_bytes(x_bytes, 'big') % self.curve.field.p

            # 计算椭圆曲线方程右边: y^2 = x^3 + ax + b (mod p)
            y_squared = ( pow(x, 3, self.curve.field.p) +
                        self.curve.a * x + self.curve.b) % self.curve.field
                            .p

            # 检查是否为二次剩余
            if  pow(y_squared, (self.curve.field.p - 1) // 2, self.curve.
                field.p) == 1:
                # 计算平方根 (适用于 p   3 (mod 4))
                y =  pow(y_squared, (self.curve.field.p + 1) // 4, self.
                    curve.field.p)

                point = Point(self.curve, x, y)
                if self._is_valid_point(point):
                    return point

            # 重试机制
            working_data += bytes([attempts % 256])
            attempts += 1

        except Exception as e:
            working_data += bytes([attempts % 256])
            attempts += 1

    raise HashToCurveError("无法生成有效点")
```

算法分析：

- 使用 try-and-increment 方法确保找到有效点

- 二次剩余判断：$y^2 \equiv a \pmod{p}$ 当且仅当 $a^{(p-1)/2} \equiv 1 \pmod{p}$
- Tonelli-Shanks 算法的简化版本（适用于 $p \equiv 3 \pmod{4}$ 的情况）
- 点验证确保生成的点确实在椭圆曲线上

4.2.4 服务器端实现

Listing 4:

```python
def _initialize_server(self, breached_credentials):
    """初始化服务器内部状态"""
    # 生成服务器私钥
    self._private_key = self.crypto.generate_private_key()

    # 预处理泄露凭据
    self._breached_prf_values = {}
    processed_count = 0

    for username, password in breached_credentials:
        try:
            # 计算PRF值
            y = self.crypto.h1(username, password)
            h2_y = self.crypto.h2_hash_to_curve(y)
            v_y_point = self._private_key * h2_y
            v_y_bytes = self.crypto.point_to_bytes(v_y_point)

            # 存储前缀以优化查询
            prefix = v_y_bytes[:self.config.PREFIX_LENGTH_BYTES]
            if prefix not in self._breached_prf_values:
                self._breached_prf_values[prefix] = []
            self._breached_prf_values[prefix].append(v_y_bytes)

            processed_count += 1
        except Exception as e:
            self.logger.warning(f"处理凭据失败: {e}")
            continue
```

设计优化：

- 预计算所有泄露凭据的 PRF 值，避免在线计算开销

- 使用前缀索引优化查询性能，避免全量比较
- 容错处理确保个别错误不影响整体初始化
- 统计信息帮助监控预处理效果

4.2.5 客户端盲化实现

Listing 5:

```python
def _perform_blinding(self) ->  dict:
    """执行盲化步骤"""
    # 计算凭据哈希
    x = self.crypto.h1(self.username, self.password)

    # 映射到曲线点
    P = self.crypto.h2_hash_to_curve(x)

    # 生成安全随机盲化因子
    t = self.crypto.generate_private_key()

    # 计算盲化点
    T = t * P
    T_bytes = self.crypto.point_to_bytes(T)

    return {
        'blinding_factor': t,
        'original_point': P,
        'blinded_point_bytes': T_bytes
    }
```

安全性分析：

- 使用密码学安全的随机数生成器生成盲化因子
- 盲化因子 t 在椭圆曲线的阶范围内均匀分布
- 盲化后的点 $T = t \cdot P$ 隐藏了原始点 P 的信息
- 返回字典便于后续去盲化操作

4.2.6 去盲化与验证

Listing 6:

```python
def _unblind_and_verify(self, server, blinding_result, server_result):
    """去盲化并验证结果"""
    # 计算盲化因子的逆元
    t = blinding_result['blinding_factor']
    t_inv =  pow(t, -1, self.crypto.curve.field.n)

    # 去盲化: V = t^(-1) * Z = t^(-1) * k * T = k * P
    Z_point = self.crypto.bytes_to_point(server_result)
    V_point = t_inv * Z_point
    V_bytes = self.crypto.point_to_bytes(V_point)

    # 前缀匹配检查
    leaked_prefixes = server.get_breached_prf_prefixes()
    my_prefix = V_bytes[:self.config.PREFIX_LENGTH_BYTES]

    if my_prefix not in leaked_prefixes:
        return False

    # 完整哈希验证避免前缀碰撞
    full_hashes = server.get_full_hashes_for_prefix(my_prefix)
    return V_bytes in full_hashes
```

验证机制：

- 模逆元计算：$t^{-1} \bmod n$，其中 n 是椭圆曲线的阶
- 两级验证：先检查前缀，再验证完整哈希
- 前缀匹配大幅减少网络传输和计算开销
- 完整验证避免前缀碰撞导致的误报

5 实验结果与分析

```
D:\Personal\Desktop\p6>python3 main.py
2025-08-10 18:58:47 - __main__ - INFO - 开始 Google Password Checkup 演示
2025-08-10 18:58:47 - __main__ - INFO - 创建包含 5 条记录的测试泄露数据库
2025-08-10 18:58:47 - __main__.PasswordCheckupServer - INFO - === [服务器] 初始化开始 ===
2025-08-10 18:58:47 - __main__.PasswordCheckupServer - INFO - [服务器] 私钥生成完成
2025-08-10 18:58:47 - __main__.PasswordCheckupServer - INFO - [服务器] 开始预处理 5 条泄露凭据
2025-08-10 18:58:48 - __main__.PasswordCheckupServer - INFO - [服务器] 预处理完成，成功处理 5 条凭据
2025-08-10 18:58:48 - __main__.PasswordCheckupServer - INFO - [服务器] 生成 5 个唯一前缀
2025-08-10 18:58:48 - __main__.PasswordCheckupServer - INFO - === [服务器] 初始化结束 ===

================================================
测试场景1: 检查已泄露的密码 ('alice', '123456')
================================================
2025-08-10 18:58:48 - __main__.PasswordCheckupClient - INFO - === [客户端] 初始化完成，用户: 'alice' ===
2025-08-10 18:58:48 - __main__.PasswordCheckupClient - INFO - [客户端] 开始检查用户 'alice' 的密码
2025-08-10 18:58:48 - __main__.PasswordCheckupClient - INFO - === [结论] 用户 'alice' 的凭据已泄露 ===
✅ 测试通过: 密码确实已泄露

================================================
测试场景2: 检查安全的密码 ('eve', 'MySecurePa$$w0rd')
================================================
2025-08-10 18:58:48 - __main__.PasswordCheckupClient - INFO - === [客户端] 初始化完成，用户: 'eve' ===
2025-08-10 18:58:48 - __main__.PasswordCheckupClient - INFO - [客户端] 开始检查用户 'eve' 的密码
2025-08-10 18:58:48 - __main__.PasswordCheckupClient - INFO - === [结论] 用户 'eve' 的凭据安全 ===
✅ 测试通过: 密码确实安全

================================================
测试场景3: 已知泄露用户的其他密码
================================================
2025-08-10 18:58:48 - __main__.PasswordCheckupClient - INFO - === [客户端] 初始化完成，用户: 'alice' ===
2025-08-10 18:58:48 - __main__.PasswordCheckupClient - INFO - [客户端] 开始检查用户 'alice' 的密码
2025-08-10 18:58:48 - __main__.PasswordCheckupClient - INFO - === [结论] 用户 'alice' 的凭据安全 ===
✅ 测试通过: 即使用户名相同，不同密码仍然安全
2025-08-10 18:58:48 - __main__ - INFO - 🎉 所有测试通过！演示完成
2025-08-10 18:58:48 - __main__ - INFO - 开始错误处理测试
2025-08-10 18:58:48 - __main__ - INFO - ✅ 空用户名检测正常
2025-08-10 18:58:48 - __main__ - INFO - ✅ 空密码检测正常
2025-08-10 18:58:48 - __main__ - INFO - ✅ 空数据库检测正常
2025-08-10 18:58:48 - __main__ - INFO - ✅ 错误处理测试完成

🎉 程序执行完成!
```

图 1

5.1 测试环境

- 操作系统：Windows 10
- Python 版本：3.x
- 关键依赖：tinyec (椭圆曲线库)
- 椭圆曲线：secp256r1 (NIST P-256)

5.2 测试数据集

实验使用了包含 5 个泄露凭据的测试数据库：

用户名	密码
alice	123456
bob	password
charlie	qwerty
david	google-sucks
eve_test	leaked_password

5.3 实验结果分析

5.3.1 服务器初始化结果

从实验输出可以看到：

- 服务器成功生成私钥 k
- 预处理了 5 条泄露凭据，全部成功
- 生成了 5 个唯一的前缀（无碰撞）
- 初始化过程无错误，系统状态正常

5.3.2 测试场景结果

场景 1：已泄露密码检测

- 输入：('alice', '123456')
- 预期结果：检测为已泄露
- 实际结果： 正确检测出密码已泄露
- 分析：客户端计算的 PRF 值与服务器预存的泄露数据匹配

场景 2：安全密码检测

- 输入：('eve', 'MySecurePa$$w0rd')
- 预期结果：检测为安全
- 实际结果： 正确检测出密码安全
- 分析：客户端计算的 PRF 值前缀不在泄露数据库中

场景 3：边界情况测试

- 输入：('alice', 'different_password')

- 预期结果：检测为安全（虽然用户名相同）
- 实际结果： 正确检测出密码安全
- 分析：证明系统能够区分同一用户的不同密码

5.3.3 错误处理测试

实验还验证了系统的错误处理能力：

- 空用户名检测正常
- 空密码检测正常
- 空数据库检测正常
- 错误处理测试完成

5.4 性能分析

5.4.1 时间复杂度

- **预处理阶段**：$O(n)$，其中 n 是泄露凭据数量
- **在线检测**：$O(\log m)$，其中 m 是唯一前缀数量
- **椭圆曲线运算**：$O(\log p)$，其中 p 是椭圆曲线的阶

5.4.2 空间复杂度

- 服务器存储：每个泄露凭据约 64 字节（椭圆曲线点）
- 客户端存储：常数空间，仅存储临时计算结果
- 通信开销：每次查询约 128 字节（两个椭圆曲线点）

6 安全性分析

6.1 隐私保护

6.1.1 客户端隐私

- **输入隐私**：服务器无法从盲化点 T 推断原始密码
- **查询隐私**：盲化因子 t 随机生成，每次查询都不同
- **结果隐私**：服务器不知道客户端的查询结果

6.1.2 服务器隐私

- **数据库隐私**：客户端只能学习自己密码的泄露状态
- **密钥保护**：服务器私钥 k 始终保密
- **统计隐私**：客户端无法推断数据库的统计信息

6.2 攻击抵抗性

6.2.1 被动攻击

- **窃听攻击**：通信内容为随机椭圆曲线点，无法破解
- **流量分析**：前缀机制隐藏了实际的查询模式
- **重放攻击**：随机盲化因子防止重放攻击

6.2.2 主动攻击

- **恶意服务器**：客户端隐私仍然得到保护
- **恶意客户端**：无法获得超出协议设计的信息
- **中间人攻击**：可通过 TLS 等传输层安全协议防护

7 实验感悟与总结

7.1 技术收获

通过本次实验，我深入理解了现代密码学在解决实际隐私保护问题中的应用：

1. **盲化技术的精妙**：盲化 PRF 协议巧妙地平衡了功能性和隐私性，允许在不泄露敏感信息的前提下进行计算。这种”计算而不知道在计算什么”的思想体现了现代密码学的深度。
2. **椭圆曲线密码学的实用性**：相比传统的 RSA 等算法，椭圆曲线密码学在相同安全级别下具有更小的密钥长度和更高的效率，这在实际部署中具有重要意义。
3. **Hash-to-Curve 的复杂性**：将任意数据安全地映射到椭圆曲线上并非 trivial，需要考虑分布均匀性、计算效率等多个因素。
4. **工程化实现的重要性**：理论上完美的协议在实现时需要考虑众多工程问题，如错误处理、性能优化、代码可维护性等。

7.2 对隐私保护的思考

1. **隐私与功能的平衡**：Google Password Checkup 展示了如何在不牺牲功能性的前提下最大化隐私保护。这种设计思路对其他隐私敏感的应用具有重要启发意义。

2. **实用性的重要性**：再好的密码学协议，如果无法高效实现或用户体验糟糕，就难以在实际中广泛应用。本协议的成功在于其简洁性和实用性。

3. **信任模型的考虑**：协议假设服务器是诚实但好奇的，这在实际部署中需要通过其他机制（如代码审计、可信执行环境等）来保证。

7.3 对密码学学习的启发

1. **理论与实践的结合**：密码学不仅仅是数学理论，更需要在实际系统中验证其可行性和安全性。

2. **安全性的多层考虑**：除了核心密码学算法的安全性，还需要考虑实现安全、协议安全、系统安全等多个层面。

3. **性能优化的艺术**：在保证安全性的前提下，如何通过巧妙的设计（如前缀匹配）来优化性能，是密码学工程实现的重要技能。

7.4 未来改进方向

1. **抗量子安全**：考虑后量子密码学算法，为未来的量子计算威胁做准备。

2. **多服务器架构**：设计分布式协议，避免单点故障和信任风险。

3. **差分隐私集成**：结合差分隐私技术，进一步增强隐私保护。

4. **批量查询优化**：支持用户一次查询多个密码，提高效率。

7.5 结论

本实验成功实现了 Google Password Checkup 的核心协议，验证了盲化 PRF 技术在隐私保护密码检测中的有效性。通过完整的系统实现和测试，我不仅掌握了相关的密码学理论，更重要的是理解了如何将理论转化为实际可用的系统。

这次实验让我深刻认识到，现代密码学已经从纯理论研究发展为解决现实世界问题的强大工具。随着隐私保护需求的日益增长，掌握这些技术对于未来的信息安全工作具有重要意义。同时，实验也提醒我们，任何技术都不是万能的，需要在特定的威胁模型和应用场景下评估其适用性和安全性。

通过本次 Project 6 的实践，我对密码学的理解从抽象的数学公式转变为具体的代码实现，这种转变不仅加深了理论理解，更培养了将复杂概念转化为实际解决方案的能力。这种能力对于未来从事信息安全相关工作具有不可估量的价值。