



Private Set Intersection

安全多方计算应用：隐私集合求交

作者：牛午甲 & 潘云石 & 石磊鑫 & 孙霄鹏 & 陈昊

组织：USTC

时间：June 9, 2023

版本：1.0



站在巨人的肩膀上。

目录

第 1 章	Abstract	1
第 2 章	Introduction	2
第 3 章	Background and Related Work	3
3.1	Background	3
3.1.1	Private set intersection (PSI)	3
3.1.2	Adversary model	3
3.1.3	Cuckoo hashing	3
3.1.3.1	插入逻辑	3
3.1.3.2	查找逻辑	3
3.1.3.3	运行时间分析	3
3.1.4	Oblivious Transfer	5
3.1.5	Garbled Circuits	7
3.2	Related Works	8
3.2.1	Are garbled circuits better than custom protocols?[huang2012private]	8
3.2.1.1	Bitwise-AND Protocol	8
3.2.1.2	Pairwise Comparisons	8
3.2.1.3	Sort-Compare-Shuffle	9
3.2.2	Efficient Batched Oblivious PRF	9
3.2.2.1	IKNP[ishai2003extending]	9
3.2.2.2	IKNP 推广 [kolesnikov2013improved]	10
3.2.3	Faster Private Set Intersection based on OT Extension[pinkas2014faster]	10
3.2.3.1	Bloom Filter-Based PSI (改进)	10
3.2.3.2	Private Set Intersection via OT	11
第 4 章	Methodology	12
4.1	Main idea	12
4.2	Blind RSA-based APSI Protocol (Version 0.1)	12
4.3	Analysis about our protocol 0.1	13
4.4	Blind RSA-based APSI Protocol (Version 1.0)	14
4.5	Analysis about our protocol 1.0	14
4.6	Blind RSA-based APSI Protocol (Version 2.0)	15
4.7	Analysis about our protocol 2.0	15
第 5 章	Experiments and Results	16
5.1	Overview	16
5.2	PSI protocol	16
5.3	APSI protocol	16
5.3.1	Normal case	16
5.3.2	Malicious case	17
第 6 章	Discussion	19

6.1	Summary	19
6.2	Future work	19

第 1 章 Abstract

隐私集合求交 (Private Set Intersection, PSI) 是一种安全多方计算的密码学技术, 它允许参与的双方在不泄露除交集之外的任何额外信息的情况下, 计算各自持有的集合的交集。这种协议可以用于各种隐私保护场景, 例如数据共享、数据分析和身份验证等。在这篇文章中, 我们将介绍隐私集合求交领域的背景和相关工作, 并对 De Cristofaro 等人提出的 PSI 协议进行编程实现。在此基础上, 我们还对 De Cristofaro 等人的 PSI 协议改进为对应的 APSI(Authorized PSI, 授权的 PSI) 版本。我们提出并设计了三个版本的基于 Blind RSA 的 APSI 协议, 分别适用于不同的时间复杂度和安全性要求。此外我们还对协议的正确性、安全性和效率进行了理论分析, 并在实际环境中进行了性能测试, 与其他现有的 PSI 协议进行了比较。我们的结果分析表明, 基于 Blind RSA 的 APSI 协议具有较高的计算效率和通信效率, 且能够抵抗非法用户的恶意攻击, 达到隐私保护和安全防御的目的。

第 2 章 Introduction

在过去的几十年里，随着信息技术的迅速发展和广泛应用，个人和组织拥有越来越多的敏感数据，例如医疗记录、金融数据等。这引发了对隐私和数据保护的关注，促使研究人员开发了各种隐私保护计算方案。

在许多场景中，两个或多个参与方可能希望计算它们的数据之间的交集，但又不希望共享原始数据，以保护隐私。例如，在医疗研究中，不同的医院可能希望比较其患者群体的共同特征，而不共享具体的患者数据；多个银行想要共享客户的风险信息，就可以使用 PSI 技术找出存在风险的客户，而不泄露其他客户的信息；运营商和金融机构想要对齐共同的客户群体，然后进行联合建模和定点营销，就可以使用 PSI 技术在不泄露客户隐私的前提下实现数据探查和模型训练等等。因此，出于隐私保护的考虑，研究人员开始研究如何在不泄露个体数据的情况下进行集合求交，即这篇文章中讨论的隐私集合求交问题，PSI。

2004 年，Freedman 等 [freedman2004efficient] 首次提出了隐私集合交集协议问题，分别构造了基于标准模型下的半诚实环境适用协议和基于随机预言机模型的恶意环境适用协议。2005 年，Kissner 等 [kissner2005privacy] 提出了多方集合协议。2010 年，De Cristofaro 等 [de2010practical] 提出了一个具有线性复杂度授权的 PSI 协议 (Authorized PSI, APSI)。2012 年，Huang 等 [huang2012private] 将 PSI 问题转化为布尔函数的电路问题。2014 年，Benny Pinkas 等 [pinkas2014faster] 提出了基于 OT 扩展的高效 PSI 协议。2016 年，Vladimir Kolesnikov 等 [kolesnikov2016efficient] 设计的基于 batched OT 扩展传输和布谷鸟哈希实现了更高效的隐私集合求交方案，成为性能上最接近朴素哈希求交技术的隐私集合求交方案。2017 年，Chen 等 [chen2017fast] 提出了基于同态加密的 PSI 协议。我们研究分析了上述文章中提出的协议思想和具体流程，分析了其方法的优点和不足。

隐私集合求交的主要挑战是如何在保证安全性和效率的同时，最小化通信开销和计算复杂度。APSI 是一种隐私集合求交的变体，它要求参与方的集合中的元素必须由一个可信的第三方（如证书颁发机构）进行授权。APSI 可以保证只有授权的元素才能参与交集计算，而未授权的元素不会泄露给对方。De Cristofaro 等人在 Practical Private Set Intersection Protocols with Linear Complexity 一文 [de2010practical] 中对以往的 APSI 协议进行改进，提出了高效的 PSI 协议。我们编写代码实现了他们的协议。

但是，De Cristofaro 等人在论文中 [de2010practical] 提及，他们设计的 PSI 协议的其中一个缺点是不知如何转为对应的 APSI 协议。为此，在 De Cristofaro 等人的基础上，进行一系列数学推导，我们设计了基于第三方证书的 APSI 协议，并给出了我们的编程实现，并进行了一系列测试。结果表明，我们的协议不仅能够完成基本的 PSI 的功能，还能对客户端的元素进行验证，防止客户端传输虚假的（携带无效证书）的元素。

第 3 章 Background and Related Work

注 这部分是我们调研的全部内容的整理与总结，包括但不限于我们在下一部分4中着重讨论的 APSI 技术，因此这部分篇幅较长。

3.1 Background

3.1.1 Private set intersection (PSI)

私有集合求交 (PSI) 是一种安全多方计算的密码学技术，它允许各自持有集合的两方比较这些集合的加密版本以计算交集。在这种情况下，除了交集的元素外，任何一方都不会向交易对手透露任何信息。形式化地，两方各自拥有集合 $S = \{s_1, s_2, \dots, s_n\}$ 和 $S' = \{s'_1, s'_2, \dots, s'_n\}$ ，其中 $s_i, s'_i \in \{0, 1\}^\sigma$ 并且假设 S, S' 中各自都没有重复的元素。目标：各方计算 $I = S \cap S'$ ，不泄露除了 I 之外的任何信息。

3.1.2 Adversary model

Semi-honest Model: 完全按照密码学的协议执行，但会尽可能尝试从收到的信息中获取额外的信息。

Malicious Model: 可以不严格按照密码学的协议执行，可以做任何事情来达到获取额外信息的目的。

3.1.3 Cuckoo hashing

[charlesCuckoo] 布谷鸟哈希最早于 2001 年由 Rasmus Pagh 和 Flemming Friche Rodler 提出。该哈希方法是了解决哈希冲突的问题而提出，利用较少计算换取了较大空间。

3.1.3.1 插入逻辑

设插入的值为 x ：

1. 若值 x 已在哈希表中则直接返回。
2. 若插入后哈希表空间不足则先进行扩容，再 rehash，然后进行 3、4、5。
3. 用哈希函数 $h_1(x)$ 计算出 x 应该插入的桶下标 i_1 ，若 $\text{bucket}[i_1]$ 空则直接插入。
4. 若 $\text{bucket}[i_1]$ 非空，则将 $\text{bucket}[i_1]$ 中的老值 x_{old} 踢开，并用哈希函数 $h_2(x_{old})$ 寻找 x_{old} 应该填的下一个位置。
5. 设插入值为 x_{old} 重复 2，直到达到最大循环次数（插入失败），或者被踢开的值都找到了新的位置（插入成功）。

3.1.3.2 查找逻辑

使用两个哈希函数 $h_1(x)$ 和 $h_2(x)$ 计算需要查找的值有可能在的两个桶，若没有匹配则查找失败，否则查找成功。

3.1.3.3 运行时间分析

插入的最坏循环次数是常数，插入的摊还时间复杂度为 $O(1)$ 。当哈希表空间不足时需要进行扩容操作，对于 n 次插入，需要 $O(\log n)$ 次扩容，每次需要进行重新哈希，大小分别为 $n, n/2, n/2^{O(\log n)}$ ，所需要的时间为 $O(n + n/2 + \dots + n/2^{O(\log n)}) < O(n + n/2 + n/4) = O(n)$ 。因此对于 n 个插入操作扩容的摊还时间为 $O(1)$ 。

1. 插入循环分析:

设插入 x 会驱逐 y , y 插入到新位置会驱逐 $z \dots$

设这个序列 $x_1 = x, x_2 = y, x_3 = z$ 为无巢序列。

情况 1: 无巢序列没有重复的元素, 即 $x_1 \neq x_2 \neq x_3$, 这种情况下经过 $O(1)$ 时间即可结束插入。

情况 2: 无巢序列中有重复元素: 这里假设 x_i 在位置 x_j 被第二次移除, $i < j$, 则此时若此位置是由 $h_1(x)$ 所得的, 则用 $h_2(x)$ 计算其下个位置, 反之亦然。这种情况下, 插入过程会回溯 x_i 前面的一系列键, 从 $x_i = x_j$ 开始, $x_{i-1} = x_{j+1}$, 直到 $x_1 = x_{j+i-1}$, 这些键都被回退到 x 未被插入之前的位置 (对应图 3.1 中 step9-12), 然后把 x 放到位置 $h_2(x)$, 这样插入过程就会继续, 直到某个 x_l 被插入一个空的位置, 或者 x_l 再次与前面的某个键相同, 如果最终陷入死循环, 则需要重新哈希。为了进行插入的时间复杂度分析, 需要使用如下引理:

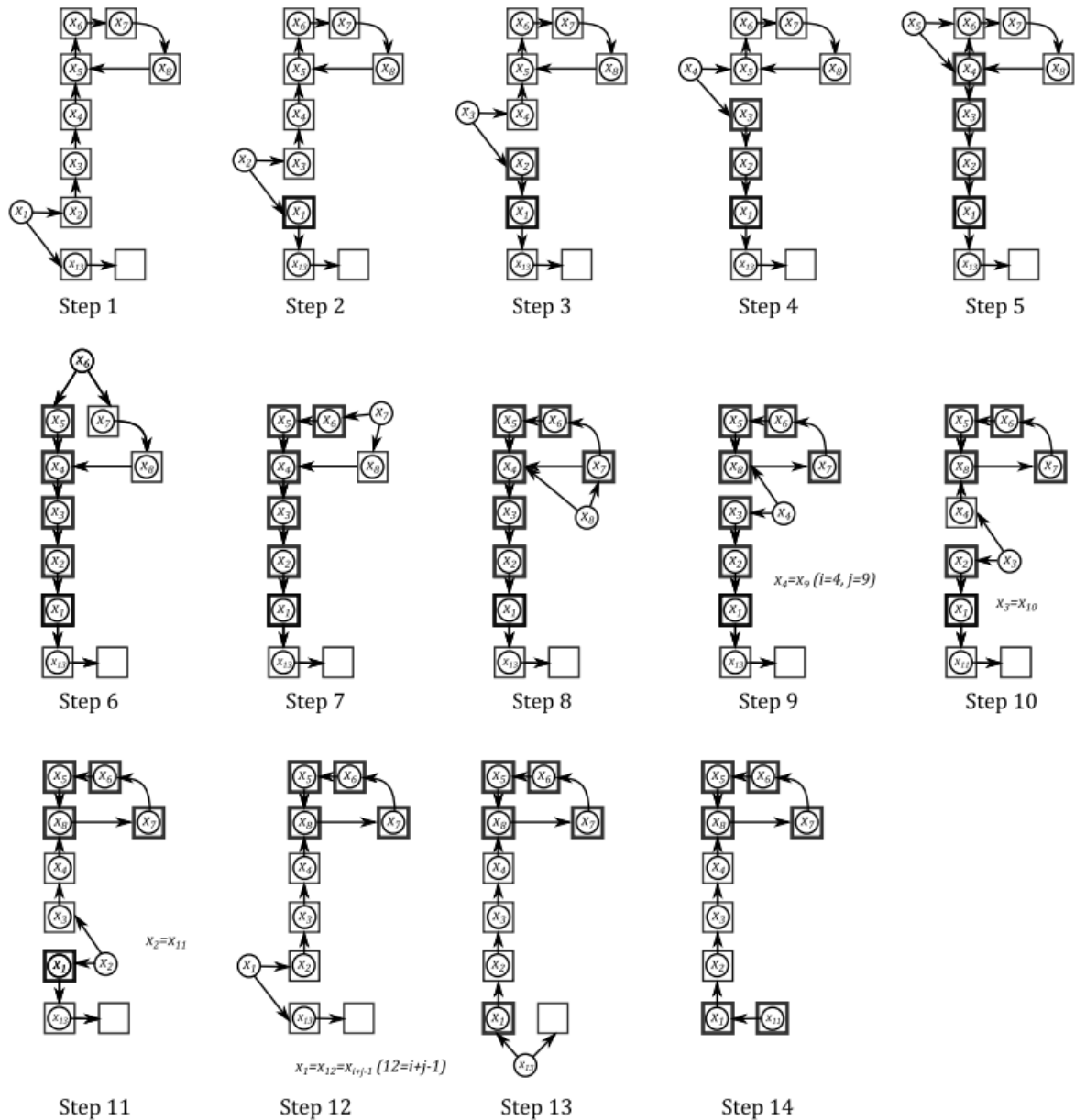


图 3.1: cuckoo hashing

引理 3.1

假设在某步 p, x 的插入过程产生一个序列 $x_1 = x, x_2, \dots, x_p$ 的无巢键, 其中还没有形成闭环 (即死循环)。在 x_1, \dots, x_p 中, 存在长度为 $l \geq p/3$ 的连续子序列 $x_q, x_{q+1}, \dots, x_{q+l}$ 其中的无巢键各不相同, 且其中 $x_q = x_1 = x$ 。♡

2. 时间复杂度分析:

考虑插入过程产生长度 $k \leq 2\text{MaxLoop}$ 的无巢键序列, 会产生以下三种情况:

情况 1: 哈希函数不是随机选则的。这种情况发生的概率为 $O(1/n^2)$ 。

情况 2: 插入过程还未陷入死循环。这种情况发生的概率为 $O(2(1+\epsilon)^{-\frac{k-1}{3}+1})$, 其中哈希表大小 $r \geq (1+\epsilon)n$ 。

情况 3: 插入过程已经陷入死循环。这种情况发生的概率为 $O(1/n^2)$ 。

证明 情况 2: 当插入过程未陷入死循环时, 根据引理 1, 有一组长度为 $v = \lceil k/3 \rceil$ 的连续不相等无巢键序列 $b_1 = x, b_2, \dots, b_v$ 。根据插入操作的方法, 连续的序列的哈希函数值相等。而哈希函数是随机选取的, 键的值域为 $0, 1, \dots, r-1$, 所以两个哈希函数的值相等的概率为 $1/r^2$, 此序列能够满足条件的概率为 $1/r^{v-1}$ 。同时, 有不多于 n^{v-1} 种产生连续不相等无巢键序列的情况, 因此产生此种序列的概率为:

$$2n^{v-1}r^{-(v-1)} = 2(r/n)^{-(v-1)} \leq 2(1+\epsilon)^{-k/3+1}$$

这里使用了条件 $r \geq (1+\epsilon)n$ 。

其他证明略。

因此, 可以计算出一个插入循环产生的无巢键数为:

$$1 + \sum_{k=2}^{2\text{MaxLoop}} [2(1+\epsilon)^{-k/3+1} + O(1/n^2)] \leq 1 + O(\text{MaxLoop}/n^2) + 2 \sum_{k=0}^{\text{inf}} (1+\epsilon)^{-k/3} = O(1) + \frac{O(1)}{1 - (1+\epsilon)^{-1/3}} = O(1)$$

因此插入迭代次数为常数, 时间复杂度为常数。

3.1.4 Oblivious Transfer

OT 协议可以让发送者在不知道接收者需要哪些信息的情况下, 互相交换信息。一般来说, OT 协议被用来在远程计算中共享私密数据, 以确保这些数据只能被授权用户获得。

OT 协议的主要思想是发送方能够向接收方发送多个消息, 但仅有其中一条消息与接收方真正需要的消息有关。发送方无法确定实际的情况, 因此不会泄露再多信息。

OT 协议可以追溯到 1981 年 Rabin 提出的方案 [rabin1981exchange], 其用途是在密钥交换中保护计算机所处网络的连续性。后来, OT 协议被证明与多方计算, 安全多方计算 (SMPC) 和私有信息检索 (PIR) 等密码学问题密切相关。更完善的不经意传输形式被称为“1-2 oblivious transfer”或“1 out of 2 oblivious transfer”, 即发送方向接收方发送两个消息, 接收方只能选择其中一个消息, 同时而言, 发送方对于接收方到底选择了哪一个消息一无所知, 由 Shimon Even、Oded Goldreich 和 Abraham Lempel 开发 [even1985randomized], 以建立用于安全多方计算的协议。后续“1-n oblivious transfer”的具体方案也被提出。

1. 第一种 OT 由 Rabin 于 1981 年 [rabin1981exchange] 提出:

其假设 Alice 有秘密 S_A , Bob 有秘密 S_B , 并且秘密为单比特, Alice 和 Bob 想要交换秘密, 要求两方都有可能得到秘密并且秘密拥有方不知道对方是否得到秘密:

- Alice 随机选取两个大素数 p, q 。并计算得到 $one-time-key\ n_A$, 然后将 n_A 发送给 Bob。
- Bob 随机选取一个数 x , 要求 $x < n_A$, 计算 $c \equiv x^2 \bmod n_A$, 然后将 c 和私钥加密的 x 发送给 Alice。
- Alice 找到一个 x_1 使得 $x_1^2 \equiv c \bmod(n_A)$, 发送 x_1 给 Bob。
- Bob 计算 $\gcd(x - x_1, n_A) = d$, 此时有 $P(d = q \text{ or } d = p) = \frac{1}{2}$ 。
- Bob 根据计算 $v_B = \begin{cases} 0, & \text{if } (x - x_1, n_A) = p \text{ or } q, \\ 1, & \text{otherwise.} \end{cases}$ 接着计算 $\varepsilon_B = v_B \oplus S_B$ 然后将 ε_B 发送给 Alice。

这是 Alice 获得 Bob 的秘密 S_B 的过程, 可以得出 Alice 得到 S_B 的概率为 $\frac{1}{2}$ 。Bob 获得 S_A 的过程依然是上述步骤, 只不过是 Alice 和 Bob 角色互换。

2. OT based on DH[naor2001efficient]

有如下假设

- (a). Alice 拥有值 v_0, v_1 和密钥 s, r_0, r_1
- (b). Bob 拥有值 $i \in 0, 1$ 和密钥 k 。Bob 想获得 v_i
- (c). Alice 和 Bob 事先统一 $g \in Z_p$, 其中 g 是大整数, p 是大素数

具体过程为:

- (a). $Alice \rightarrow Bob: g^s$.
- (b). Bob 基于 i 生成

$$L_i = \begin{cases} g^k & , \text{if } i = 0, \\ g^{s-k} & , \text{if } i = 1. \end{cases}$$

- (c). $Bob \rightarrow Alice: L_i$
- (d). Alice 生成 C_0, C_1
 $C_0 = (g^{r_0}, (L_i)^{r_0} \oplus v_0)$
 $C_1 = (g^{r_1}, (g^s/L_i)^{r_1} \oplus v_1)$
- (e). $Alice \rightarrow Bob: C_0, C_1$
- (f). Bob 解密 v_i

I. case $i = 0$

A. Bob 可以通过如下方式解密获得 v_0 :

$$C_0[0]^k \oplus C_0[1] = (g^{r_0})^k \oplus (L_i)^{r_0} \oplus v_0 = (g^{r_0})^k \oplus (g^k)^{r_0} \oplus v_0 = v_0$$

B. Bob 无法获得 v_1 因为 $C_1[1] = (g^s/L_i)^{r_1} \oplus v_1 = g^{(s-k)r_1} \oplus v_1$ 而 Bob 不知道 s, r_1

II. case $i = 1$

A. 类似地, Bob 通过如下方式解密获得 v_1 :

$$C_1[0]^k \oplus C_1[1] = (g^{r_1})^k \oplus (g^s/L_i)^{r_1} \oplus v_1 = (g^{r_1})^k \oplus (g^k)^{r_1} \oplus v_1 = v_1$$

B. Bob 无法获得 v_0 因为 $C_0[1] = (L_i)^{r_0} \oplus v_0 = g^{(s-k)r_0} \oplus v_0$ 而 Bob 不知道 s, r_0

III. 因此, Bob 只能解密 v_i 而不能解密 v_{1-i}

3. OT based on RSA[OTRSA]

Even、Goldreich 和 Lempel 的协议可以使用 RSA 加密方案来实现, 如下:

- (a). Alice 拥有消息 m_0, m_1 , 并其中一个发送给 Bob。Bob 不想让 Alice 知道他收到了哪一个。
- (b). Alice 生成一个 RSA 密钥对, 包括模数 N , 公有指数 e 和私有指数 d 。
- (c). Alice 还生成了两个随机值 x_0, x_1 并将它们与模数 N 和指数 e 一起发送给 Bob
- (d). Bob 选择 $b \in \{0, 1\}$ 并生成随机数 k ; 随后计算 $v = (x_b + k^e) \bmod N$, 并将 v 发送给 Alice。
- (e). Alice 计算 $k_0 = (v - x_0)^d \bmod N$ 和 $k_1 = (v - x_1)^d \bmod N$; 于是 k_0 和 k_1 其中之一是 k , 但 Alice 并不知道是哪一个。
- (f). Alice 计算 $m'_0 = m_0 + k_0$ 和 $m'_1 = m_1 + k_1$, 将其发送给 Bob。
- (g). Bob 计算 $m_b = m'_b - k$, 于是 Bob 获得了 Alice 发送的消息 m_b 。

3.1.5 Garbled Circuits

混淆电路协议分为四个部分。

1. Alice 生成混淆电路

生成过程主要分四步：

- (a). Alice 对电路中的每一线路（Wire）进行标注。如下图所示，Alice 一共标注了七条线路，包括模块的输入输出 $W_{a_0}, W_{b_0}, W_{c_0}, W_{c_1}$ ，和模块内的中间结果 W_d, W_e, W_f 。对于每一条线路 W_i ，Alice 生成两个长度为 k 的字符串 X_i^0, X_i^1 。这两个字符串分别对应逻辑上的 0 和 1。这些生成的标注会在 Step 2 有选择性地发给 Bob，但 Bob 并不知道 X_i^0, X_i^1 对应的逻辑值。

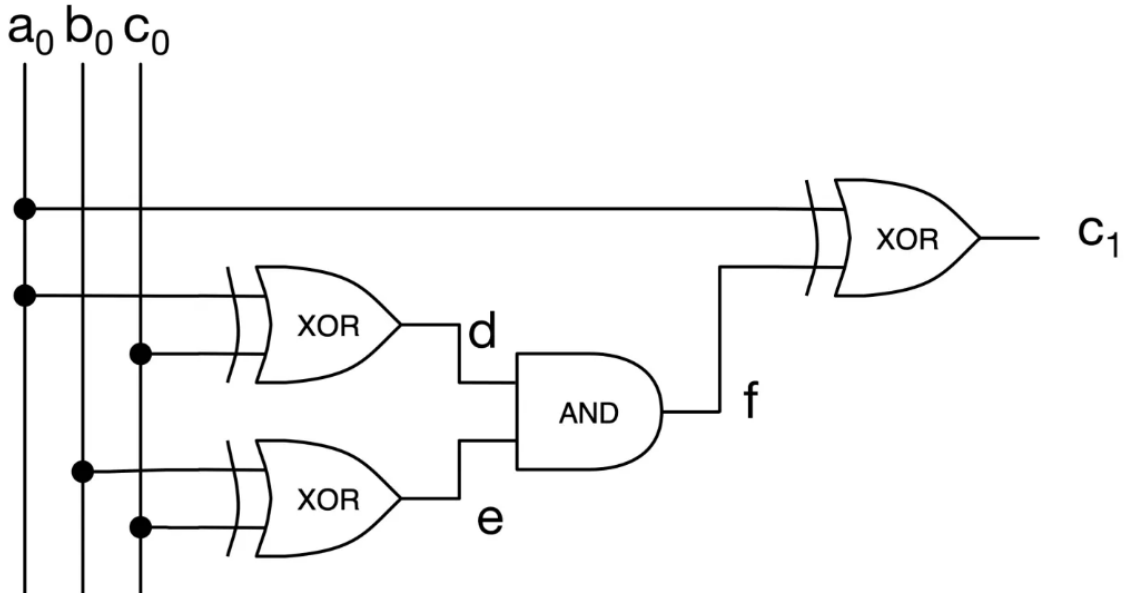


图 3.2: circuit

- (b). Alice 对电路中的每一个逻辑门的 Truth Table 用 X_i^0, X_i^1 进行替换，由 X_i^0 替换 0，由 X_i^1 替换 1。
- (c). Alice 对每一个替换后的 Truth Table 的输出进行两次对称密钥加密（即加密和解密的密钥相同），加密的密钥是 Truth Table 对应行的两个输入。比如 Truth Table 的第一行是 $X_{a_0}^0, X_{c_0}^0, X_d^0$ ，我们就用 $X_{a_0}^0, X_{c_0}^0$ 加密 X_d^0 生成 $\text{Enc}(X_{a_0}^0, X_{c_0}^0)X_d^0$ 。
- (d). Alice 对第三步加密过后的 Truth Table 的行打乱得到 Garbled Table。所以 Garbled Table 的内容和行号就无关了。混淆电路的混淆二字便来源于这次打乱。

2. Alice 和 Bob 通信

- (a). Alice 将她的输入对应的字符串发送给 Bob。比如 $a_0 = 1$ ，那 Alice 会发送 $X_{a_0}^1$ 给 Bob。由于 Bob 不知道 $X_{a_0}^1$ 对应的逻辑值，也就无从知晓 Alice 的秘密了。
- (b). Bob 通过不经意传输 (OT) 协议从 Alice 获得他的输入对应的字符串。不经意传输保证了 Bob 在 $X_{b_0}^0, X_{b_0}^1$ 中获得一个，且 Alice 不知道 Bob 获得了哪一个。所以 Alice 也就无从知晓 Bob 的 b_0 了。
- (c). 最后，Alice 将所有逻辑门的 Garbled Table 都发给 Bob。在这个例子中，一共有四个 Garbled Table。

3. Bob evaluate 生成的混淆电路

Alice 和 Bob 通信完成之后，Bob 便开始沿着电路进行解密。

因为 Bob 拥有所有输入的标签和所有 Garbled Table，他可以逐一对每个逻辑门的输出进行解密。在这个例子中，假设 Bob 拥有的输入标签为 $X_{a_0}^1, X_{b_0}^1, X_{c_0}^0$ 。他可以：

对于电路图左上方的 XOR，用 $X_{a_0}^1, X_{c_0}^0$ 解密获得 X_d^1 ；

对于电路图左下方的 XOR，用 $X_{b_0}^1, X_{c_0}^0$ 解密获得 X_e^1 ；

对于电路图中间的 AND，用 X_d^1, X_e^1 解密获得 X_f^1 ；

对于电路图右侧的 XOR，用 X_{a0}^1, X_f^1 解密 X_{c1}^0 ：

值得注意的是，由于 Garbled Table 每一行的密钥都不同，所以 Bob 只能解密其中一行。而且 Bob 并不知道解密出来的 $X_d^1, X_e^1, X_f^1, X_{c1}^0$ 对应的逻辑值，也就无从获得更多信息了。而 Alice 全程不参与 Bob 的解密过程，所以也如法获得更多信息。

4. 共享结果

最后 Alice 和 Bob 共享结果。Alice 分享 X_{c1}^0, X_{c1}^1 或者 Bob 分享 X_{c1}^1 ，双方就能获得电路输出的逻辑值了。

3.2 Related Works

3.2.1 Are garbled circuits better than custom protocols?[huang2012private]

这篇文章 [huang2012private] 提出了 3 个基于 Yao's generic garbled circuit 来执行的 PSI 协议。使用这样的技术生成隐私保护协议有几个优点：

1. 通过依赖现有的软件包来构建乱码电路协议，人们只需要为要计算的功能编写一个电路，而不必从头开始设计和实现一个新的协议。
2. 通用协议本身也比定制协议更加模块化。

3.2.1.1 Bitwise-AND Protocol

适用场景：universe 很小

主要思想：集合可以用一个长为 2^σ 的 0-1 向量表示，集合的交集可以通过对双方的 0-1 向量进行按位与来计算。输出恰好是交集的 0-1 向量表示。

计算电路： 2^σ 个二进制与门即可。

缺点：

最终协议的开销随着 σ 呈指数增长。

BWA 协议不限制各方集合的大小，因此不诚实的参与者可以使用全 1 向量作为输入，从而了解其他参与者的整个集合。

3.2.1.2 Pairwise Comparisons

PWC 时间复杂度： $O(\sigma n^2)$

只有判断相等的 'Equal' 函数需要 garbled circuit 实现。具体地将两个 σ bits 的输入按位异或，在将所有位求位或即可。

算法会在找到匹配项后立即显示匹配项

优点：

这可以提高性能，避免对任何已经匹配的元素执行进一步的比较。

性能提升的理论分析：设集合 S 和 S' 大小都是 n ，交集的大小为 \hat{n} ，由于 S 和 S' 是随机排列的，所以这 \hat{n} 个相交元素期望分布是均匀的，将集合划分为 $\hat{n} + 1$ 个区间，每个区间有 $\frac{n-\hat{n}}{\hat{n}+1}$ 个元素，对于第 i ($0 \leq i \leq \hat{n}$) 个区间中的每个元素，都要执行 $n - i$ 次比较；对于第 i ($0 \leq i \leq \hat{n} - 1$) 个相交的元素，平均意义下需要执行 $\frac{n-i}{2}$ 次比较。因此，总共比较次数：

$$N_{Equal} \approx \sum_{i=0}^{\hat{n}} \frac{(n - \hat{n})(n - i)}{\hat{n} + 1} + \sum_{i=0}^{\hat{n}-1} \frac{n - i}{2} = \frac{(2n - \hat{n})^2 + \hat{n}}{4}.$$

缺点:

可能会泄露有关各方集合中元素的位置信息, 因为参与方了解找到匹配元素的顺序。解决方法是各方在开始协议之前随机排列其集合。

大大增加了通信复杂度, 因为每个 reveal 操作都增加了额外的一轮通信。解决方法是使用两个线程实现协议, 其中 reveal 是异步完成的, 而主线程比较每个可能的元素对。一旦 reveal 线程找到了匹配项, 主线程就会被异步通知跳过涉及匹配元素的所有不必要的比较。

3.2.1.3 Sort-Compare-Shuffle

SCS 时间复杂度: $O(n \log n)$

主要思想: 每一方都首先对其集合进行本地排序。然后双方实现一个 oblivious merging network, 利用两个输入集都已排序的事实, 对它们集合的并集进行排序。接下来使用 garbled circuits 来比较排序序列中的相邻元素以找到所有匹配项。但是, 在此阶段直接输出匹配项将揭示不在交集集中的元素的信息。(例如, 如果双方得知排序列表中的前两个元素匹配, 这将向第一方透露, 第二方的集合不包含任何小于第一个匹配元素的元素) 因此需要对匹配元素的列表进行洗牌, 这样就不会显示匹配元素的位置。

3.2.2 Efficient Batched Oblivious PRF

本文介绍了 IKNP 协议 [ishai2003extending] 以及其改进协议 KK[kolesnikov2013improved]:

IKNP[ishai2003extending]: 一个非常高效的 OT 协议, 基于 Yao's Garbled Circuit (GC) 实现。

KK[kolesnikov2013improved]: 是 IKNP 协议在处理短秘密上的一个改进。它将 1-out-of-n OT 的开销降到接近 1-out-of-2 OT 的开销 ($n \leq 256$)。

3.2.2.1 IKNP[ishai2003extending]

目标: 使用 k 次 1-out-of-2 OT 实现实际上 $m \gg k$ 次 1-out-of-2 OT。

Notation: \mathbf{t}_j 表示矩阵 T 的第 j 行; \mathbf{t}^j 表示矩阵 T 的第 j 列。 \mathbf{u}_j 和 \mathbf{u}^j 同理。

receiver: $r \in \{0, 1\}^m, T \in \{0, 1\}^{m \times k}, U \in \{0, 1\}^{m \times k}, s.t. \mathbf{t}_j \oplus \mathbf{u}_j = r_j \cdot \mathbf{1}^k$

sender: $s \in \{0, 1\}^k$

具体过程

1. sender 和 receiver 角色互换进行 k 次 1-out-of-2 OT: 在每一次 OT 中, receiver 向 sender 发送 \mathbf{t}^i 和 \mathbf{u}^i , sender 根据自己拥有的 s_i 来决定接收 \mathbf{t}^i 或 \mathbf{u}^i 。 k 次 OT 后, sender 将选择的输出拼成矩阵 $Q \in \{0, 1\}^{m \times k}$, 并且可以证明:

$$\mathbf{q}_j = \mathbf{t}_j \oplus [r_j \cdot s] = \begin{cases} \mathbf{t}_j & \text{if } r_j = 0 \\ \mathbf{t}_j \oplus s & \text{if } r_j = 1 \end{cases}$$

一个简要证明如下: 由上述过程, 结合 $\mathbf{t}_j \oplus \mathbf{u}_j = r_j \cdot \mathbf{1}^k$ 可以得知:

$$\mathbf{q}_j^i = \begin{cases} \mathbf{t}_j^i & \text{if } s_i = 0 \\ \mathbf{u}_j^i = \mathbf{t}_j^i \oplus r_j & \text{if } s_i = 1 \end{cases}$$

表达成行向量的形式, 显然有 $\mathbf{q}_j = \mathbf{t}_j \oplus [r_j \cdot s]$ 。

2. 如上操作之后, sender 拥有 m 个消息对:

$(\mathbf{t}_j, \mathbf{t}_j \oplus s) = (\mathbf{q}_j, \mathbf{q}_j \oplus s), 1 \leq j \leq m$, receiver 拥有 \mathbf{t}_j 。这就通过 k 次 OT 完成了实际上的 $m(\gg k)$ 次 OT。由于 receiver 并不知道 s , 所以 receiver 无法得知 $\mathbf{t}_j \oplus s$; 由于 sender 不知道 r , 而 \mathbf{q}_j 和 \mathbf{t}_j 、 $\mathbf{t}_j \oplus s$ 二者中哪个相等完全由 r 决定, 因此 sender 不知道 receiver 拥有消息对中的哪一条消息。因此这是满足 OT 定义的。

存在的问题与改进

1. 上述的消息对之间重复的使用了同一个比特串 s ，这使得生成的消息之间存在相关性，所以必须解决这种相关性。IKNP 协议采用的方法是：使用一个 Random Oracle H_i 来解决相关性，即将消息对哈希，成为 $(H_i(q_i), H_i(q_i \oplus s))$ ，Bob 的消息为 $H_i(t_i)$ ，从而解决上述问题。
2. 使用 k 次 OT 实现实际上 $m \gg k$ 次 OT 会使每次 OT 的消息变长。解决方法是：对消息对中两个长信息分别使用不同公钥加密，然后使用 OT 传对应的私钥。

3.2.2.2 IKNP 推广 [kolesnikov2013improved]

目标：实现 1-out-of- 2^l OT。

主要思想：将上述 1-out-of-2 IKNP OT 过程中的 choice bit r_i 换成一个 l -bit string. C 是 l 的 k -bit 线性纠错码。

具体过程：

receiver 生成的 $T \in \{0, 1\}^{m \times k}, U \in \{0, 1\}^{m \times k}$ 满足的约束条件变为 $\mathbf{t}_j \oplus \mathbf{u}_j = C(r_j)$. 在相同操作下，sender 得到的矩阵也相应变为 $\mathbf{q}_j = \mathbf{t}_j \oplus [C(r_j) \cdot s]$ ，其中 \cdot 是按位与。sender 无法得知 r_j 以及 $C(r_j)$ ，因此 sender 可以遍历 r_j 的全部 2^l 个可能取值，计算出 2^l 个 $H(\mathbf{q}_j \oplus [C(r_j) \cdot s])$. receiver 只拥有其中一个，即 $H(\mathbf{t}_j)$ 。

由于 receiver 并不知道 s ，所以 receiver 无法得知 $\mathbf{t}_j \oplus [C(r') \cdot s], r' \in \{0, 1\}^l$ ；由于 sender 不知道 r_j ，而 \mathbf{q}_j 和 $\mathbf{t}_j \oplus [C(r') \cdot s], r' \in \{0, 1\}^l$ 中哪个相等完全由 r_j 决定，因此 sender 不知道 receiver 拥有消息对中的哪一条消息。因此这是满足 OT 定义的。

这完成了 random strings 的 OT。对于 chosen strings 的 OT，发送方只需要将 2^l 个 chosen strings 分别使用 2^l 个不同的 $H(\mathbf{q}_j \oplus [C(r') \cdot s]), r' \in \{0, 1\}^l$ 作为对称密钥加密后发给 receiver 即可。

安全性保证：

假设 receiver 拥有 r_j ，还想知道 2^l 个字符串中其他的字符串 $H(\mathbf{q}_j \oplus [C(\tilde{r}) \cdot s])$ ，注意到：

$\mathbf{q}_j \oplus [C(\tilde{r}) \cdot s] = \mathbf{t}_j \oplus [C(r_j) \cdot s] \oplus [C(\tilde{r}) \cdot s] = \mathbf{t}_j \oplus [(C(r_j) \oplus C(\tilde{r})) \cdot s]$ ，显然只有 s 是 receiver 未知的，因此如果 receiver 采用穷举攻击的方法，穷举空间大小取决于 $C(r_j) \oplus C(\tilde{r})$ 的 Hamming weight (即 1 的个数)。

注 对于上面用到的线性纠错码 C 的讨论。

C 不需要具有纠错码的许多特性：

1. 不需要解码
2. 安全性由 $C(r_j) \oplus C(\tilde{r})$ 的 Hamming weight 保证

因此，可以将对 C 的要求从纠错码放宽到伪随机码 (pseudorandom code, PRC)，消除了对 receiver 选择字符串大小的限制。 $C : \{0, 1\}^* \rightarrow \{0, 1\}^l$

3.2.3 Faster Private Set Intersection based on OT Extension[pinkas2014faster]

本文改进了两个已经存在的 PSI 协议，并提出了一种基于 OT 的高效 PSI 协议。由于 SCS 已经在 3.2.1 节介绍，这里就不再进行介绍。

3.2.3.1 Bloom Filter-Based PSI (改进)

The Bloom Filter

本质上是一个很长的二进制向量和一系列随机映射函数。Bloom Filter 可以用于检索一个元素是否在一个集合中。

Garbled Bloom Filter-Based PSI

对于基于 BF 的 PSI，不能简单地计算代表每个集合的 BF 的按位与，因为这会泄露信息。(C. Dong Et al. 2013)

改进：Garbled Bloom Filter (GBF)：仍然是 κ 个哈希函数，但是 $G[i]$ 从单 bit 变成了一个长度为 l 的 share。元素 x 在集合中 $\iff \bigoplus_{j=1}^{\kappa} G[h_j(x)] = x$

由于现有的 share 需要被重用, GBF 的生成不能完全并行化。

Random GBF

核心理念是让各方协作生成随机 GBF。原始协议中, GBF 必须具有特定的结构 ($\oplus_{j=1}^n G[h_j(x)] = x$)。修改后的协议可以基于随机 OT 扩展。对于滤波器中的每个位置, 如果其 BF 中对应的位为 1, 则每一方学习一个随机值。然后 P1 将每个输入对应的 GBF 值的异或发送给 P2, P2 将这些值与自己输入的 GBF 值的异或进行比较。

oblivious pseudo-random generator (OPRG)

从每个参与方 P_i 接收输入 $b_i \in \{0, 1\}$, 生成一个 random string s , 如果 $b_i = 1$ 就把 s 发给 P_i , 否则什么都不发。要求参与方不知道对方是否获得了 s 。

random OT extension: S 在第 i 个 OT 中没有输入, 输出两个值 (x_0^i, x_1^i) , 而 R 输入一个选择位向量 \mathbf{b} , 输出 $x_{b[i]}^i$ 。新的功能是通过让 S 忽略它接收到的 x_0^i 输出, 并且如果 $b_1 = 0$ 也忽略 x_1^i 输出; 类似地, 如果 $b_2 = 0$, R 忽略它的输出。random OT extension 协议因此变得更高效, 因为各方可以忽略部分计算。

OPRG 本质上是个 GBF 的生成器

3.2.3.2 Private Set Intersection via OT

一种新的私有集交集协议, 该协议基于最高效的 OT 扩展技术, 特别是 random OT 和高效的 $1-out-of-n$ OT。随着集合大小的增加, 这种 PSI 协议可以非常有效地扩展。

The Basic PEQT Protocol

改进: 将输入使用 N 进制表示, 只需要一次 $1-out-of-n$ OT。

原理: $\sigma-bit$ 二进制最大 2^σ , N 进制下的位数: $t = \log_N 2^\sigma = \sigma \log_N 2$, 论文中 $N = 2^\eta$, 故 $t = \frac{\sigma}{\eta}$ 。

Private Set Inclusion Protocol

为了并行地执行多个比较, 需要在更长的字符串上计算 OTs, 本质上是 (并行地) 为集合 X 中的每个元素传输一个随机字符串。

The OT-Based PSI Protocol: P_2 调用 Private Set Inclusion Protocol n_2 次即可。

第 4 章 Methodology

4.1 Main idea

我们希望在 Blind RSA-based PSI Protocol [de2010practical] 的基础上，给 Client 集合中的元素加上签名，以防止 Client 通过伪造数据的方法得到 Server 集合中的数据。

问题的主要难点在于 Blind RSA-based PSI Protocol 中已经存在了一组 RSA 公私钥 (n, e, d) ，而如果让 CA 给 Client 集合中的元素签名，也需要一组 RSA 公私钥。这两组 n 是不能相同的，因为 d 的生成依赖于 n 的素因子 p, q 。如果 CA 和 Server 使用相同的 n ，那么由于 n 有且仅有两个素因子，为了生成 d ，CA 和 Server 都需要知道 (p, q) 。这将导致双方可以轻易地解密对方加密的信息，带来安全问题。因此这两组 RSA 公私钥中的 n 不能相同，不妨设 CA 发布的公钥是 n_1, e_1 ，私钥是 d_1 ，Server 发布的公钥是 n_2, e_2 ，私钥是 d_2 。由此引发新的问题：原始协议是在 \mathbb{Z}_n 中计算的，现在存在不同的 n_1, n_2 ，应该在何种代数结构下计算呢？

针对上面的问题，我们主要的想法是利用引理 4.1。因此，为了在协议过程中同时保留在 \mathbb{Z}_{n_1} 和 \mathbb{Z}_{n_2} 中的可计算性，我们不妨在 \mathbb{Z}_N 中计算，其中 $N = [n_1, n_2]$ 。由于 n_1, n_2 各自都只有两个很大的素因子： p_1, q_1 和 p_2, q_2 ，因此可以认为 $(n_1, n_2) = 1$ ，从而 $N = [n_1, n_2] = n_1 n_2$ 。至此我们给出我们初始版本的协议。

4.2 Blind RSA-based APSI Protocol (Version 0.1)

Blind RSA-based APSI Protocol (Version 0.1)

- Common input: $n_1, e_1, n_2, e_2, H(), H'(), N = n_1 n_2$

- CA's input: d_1

- Client's input: $\mathcal{C} = \{hc_1, hc_2, \dots, hc_v\}$, where: $hc_i = H(c_i)$

- Server's input: $d_2, \mathcal{S} = \{hs_1, hs_2, \dots, hs_w\}$, where: $hs_j = H(s_j)$

- 1 : Server: Calculate $\forall j, K_{s:j} = hs_j^{d_2} \bmod N$
- 2 : Client: Send $\{hc_1, \dots, hc_v\}$ to CA
- 3 : CA: Sign $\{hc_1, \dots, hc_v\} : \forall i, \sigma_i = (hc_i)^{d_1} \bmod N$
- 4 : CA: Send $\{\sigma_1, \sigma_2, \dots, \sigma_v\}$ to Client
- 5 : Client: $\forall i, R_{c:i} \leftarrow \mathbb{Z}_N^*, x_i = \sigma_i R_{c:i}^{e_1} \bmod N, y_i = \sigma_i R_{c:i}^{e_2} \bmod N, z_i = \sigma_i R_{c:i}^{e_1 e_2} \bmod N$
- 6 : Client: Send $x_1, x_2, \dots, x_v, z_1, z_2, \dots, z_v$ to CA, send y_1, y_2, \dots, y_v to server
- 7 : CA: Calculate $x'_i = x_i^{d_1} \bmod N, z'_i = z_i^{d_1} \bmod N, \forall i$
- 8 : CA: Send $x'_1, x'_2, \dots, x'_v, z'_1, z'_2, \dots, z'_v$ to client
- 9 : Server: Calculate $y'_i = y_i^{d_2} \bmod N, \forall i$ and $t'_j = H'(K_{s:j}^{(1-e_1)} \bmod n_1), \forall j$
- 10 : Server: Send $y'_1, y'_2, \dots, y'_v, t'_1, t'_2, \dots, t'_w$ to client
- 11 : Client: Send z'_1, z'_2, \dots, z'_v to server
- 12 : Server: Calculate $z''_i = z'_i^{d_2} \bmod N, \forall i$
- 13 : Server: Send $z''_1, z''_2, \dots, z''_v$ to client
- 14 : Client: Calculate $t_i = H'((z''_i x'^{-1}_i y'^{-1}_i R_{c:i}^{-1})^{e_1} hc_i \bmod n_1), \forall i$
- 15 : Client OUTPUT: $\{t_1, t_2, \dots, t_v\} \cap \{t'_1, t'_2, \dots, t'_w\}$

4.3 Analysis about our protocol 0.1

在证明该协议的正确性之前，需要先证明两个引理：

引理 4.1

若 $m|n$ ，则 $x \equiv y \pmod{n} \implies x \equiv y \pmod{m}$



引理 4.2

$\forall x \in \mathbb{Z}_N, x^{e_1 e_2 d_1 d_2 - e_1 d_1 - e_2 d_2} \equiv x \pmod{N}$ ，其中 e_1, e_2, d_1, d_2, N 定义如前面所述。



证明 由于 $N = n_1 n_2$ ，其中 $n_1 = p_1 q_1, n_2 = p_2 q_2$ 且 p_1, q_1, p_2, q_2 为两两不同的素数，因此 $\varphi(N) = (p_1 - 1)(p_2 - 1)(q_1 - 1)(q_2 - 1)$ ， $\varphi(n_1) = (p_1 - 1)(q_1 - 1)$ ， $\varphi(n_2) = (p_2 - 1)(q_2 - 1)$ 。由 RSA 中私钥的生成过程，有 $e_1 d_1 \equiv 1 \pmod{\varphi(n_1)}$ ， $e_2 d_2 \equiv 1 \pmod{\varphi(n_2)}$ 。从而 $(e_1 d_1 - 1)(e_2 d_2 - 1) \equiv 0 \pmod{\varphi(N)}$ ，即 $e_1 e_2 d_1 d_2 - e_1 d_1 - e_2 d_2 \equiv 1 \pmod{\varphi(N)}$ 。由 Euler 定理可得 $x^{e_1 e_2 d_1 d_2 - e_1 d_1 - e_2 d_2} \equiv x^{k\varphi(N)+1} \equiv x \pmod{N}$ 。

该协议的正确性见如下定理：

定理 4.1

执行完 4.2 协议后，Client 能够得到自己集合和 Server 集合的交集元素。



证明 执行完协议的前 5 步之后，Client 方拥有 $x_i = \sigma_i R_{c:i}^{e_1} \pmod{N}$ ， $y_i = \sigma_i R_{c:i}^{e_2} \pmod{N}$ ， $z_i = \sigma_i R_{c:i}^{e_1 e_2} \pmod{N}$ ；执行完协议的前 10 步之后，Client 方拥有 $x'_i = \sigma_i^{d_1} R_{c:i}^{e_1 d_1} \pmod{N}$ ， $y'_i = \sigma_i^{d_2} R_{c:i}^{e_1 d_2} \pmod{N}$ ， $z'_i = \sigma_i^{d_1} R_{c:i}^{e_1 e_2 d_1} \pmod{N}$ ， $K_{s:j} = h s_j^{d_2} \pmod{N}$ ；执行完协议的前 13 步之后，Client 方拥有 $x'_i = \sigma_i^{d_1} R_{c:i}^{e_1 d_1} \pmod{N}$ ， $y'_i = \sigma_i^{d_2} R_{c:i}^{e_1 d_2} \pmod{N}$ ， $z''_i = \sigma_i^{d_1 d_2} R_{c:i}^{e_1 e_2 d_1 d_2} \pmod{N}$ ， $K_{s:j} = h s_j^{d_2} \pmod{N}$ 。因此，协议第 14 步中 Client 的计算 $t_i = (z''_i x'^{-1}_i y'^{-1}_i)^{e_1} h c_i = (\sigma_i^{d_1 d_2 - d_1 - d_2} R_{c:i}^{e_1 e_2 d_1 d_2 - e_1 d_1 - e_2 d_2 - 1})^{e_1} h c_i = h c_i^{e_1^2 d_1^2 d_2^2 - e_1^2 d_1^2 - e_1^2 d_1 d_2 + 1} \pmod{N} = h c_i^{d_2(1-e_1)} \pmod{n_1}$ ，将此结果与 $K_{s:j}^{(1-e_1)} = h s_j^{d_2(1-e_1)} \pmod{n_1}$ 对比即可求出交集元素。

然而这个协议是不安全的。具体地，我们发现了如下攻击方法：

命题 4.1

Client 可以通过伪造元素来欺骗 Server 从而获得 Server 集合中的一些数据。具体地，Client 伪造数据 $h c_{fake}$ ，将其混入 $\sigma_1, \sigma_2, \dots, \sigma_v$ ，然后正常执行协议的 5-13 步，在 14 步中 t_{fake} 的计算改为 $t_{fake} = H'((z''_{fake} x'^{-1}_{fake} y'^{-1}_{fake} R_{c:fake}^{-1})^{e_1} h c_{fake} \pmod{n_1})$ 即可。



注 仔细分析协议的流程可以发现，造成上述攻击方法可行的原因在于最终的签名没有由 Server 验证，而是 Client 自己在最后解除了签名。解决此问题的思路主要有两种：

1. 在协议的最终验证阶段，Server 想办法让 Client 无法直接解除自己集合元素的签名。
2. 在协议的过程中，加入 Server 验证 Client 集合元素签名的过程。

基于这两个不同的出发点，我们提出了协议 1.0 版本 4.4 与协议 2.0 版本 4.6

4.4 Blind RSA-based APSI Protocol (Version 1.0)

Blind RSA-based APSI Protocol (Version 1.0)

- Common input: $n_1, e_1, n_2, e_2, H(), H'(), N = n_1 n_2$
- CA's input: d_1
- Client's input: $\mathcal{C} = \{hc_1, hc_2, \dots, hc_v\}$, where: $hc_i = H(c_i)$
- Server's input: $d_2, \mathcal{S} = \{hs_1, hs_2, \dots, hs_w\}$, where: $hs_j = H(s_j)$

- 1 : Server: Calculate $\forall j, K_{s:j} = hs_j^{d_2} \bmod N$
- 2 : Client: Send $\{hc_1, \dots, hc_v\}$ to CA
- 3 : CA: Sign $\{hc_1, \dots, hc_v\} : \forall i, \sigma_i = (hc_i)^{d_1} \bmod N$
- 4 : CA: Send $\{\sigma_1, \sigma_2, \dots, \sigma_v\}$ to Client
- 5 : Client: $\forall i, R_{c:i} \leftarrow \mathbb{Z}_N^*, x_i = \sigma_i R_{c:i}^{e_1} \bmod N, y_i = \sigma_i R_{c:i}^{e_2} \bmod N, z_i = \sigma_i R_{c:i}^{e_1 e_2} \bmod N$
- 6 : Client: Send $x_1, x_2, \dots, x_v, z_1, z_2, \dots, z_v$ to CA, send y_1, y_2, \dots, y_v to server
- 7 : CA: Calculate $x'_i = x_i^{d_1} \bmod N, z'_i = z_i^{d_1} \bmod N, \forall i$
- 8 : CA: Send $x'_1, x'_2, \dots, x'_v, z'_1, z'_2, \dots, z'_v$ to client
- 9 : Server: Calculate $y'_i = y_i^{d_2} \bmod N, \forall i$ and $t'_{i:j} = H'(y_i^{e_1} K_{s:j}^{(1-e_1)} \bmod n_1), \forall j$
- 10 : Server: Send $t'_{1:1}, t'_{1:2}, \dots, t'_{v:w}$ to client
- 11 : Client: Send z'_1, z'_2, \dots, z'_v to server
- 12 : Server: Calculate $z''_i = z'_i^{d_2} \bmod N, \forall i$
- 13 : Server: Send $z''_1, z''_2, \dots, z''_v$ to client
- 14 : Client: Calculate $t_i = H'((z''_i x_i'^{-1} R_{c:i}^{-1})^{e_1} hc_i \bmod n_1), \forall i$
- 15 : Client OUTPUT: $\{t_1, t_2, \dots, t_v\} \cap \{t'_{1:1}, t'_{1:2}, \dots, t'_{v:w}\}$

4.5 Analysis about our protocol 1.0

Correctness 1.0 版本协议的原理和 0.1 版本的原理完全相同，因此其正确性可由 0.1 版本协议的正确性 4.1 保证。

Security 相比于协议 0.1 版本 4.2，这一版本的协议主要在第 9 步和第 14 步做了修改。这种修改将使 Client 无法通过之前的方法 4.1 进行攻击，因为 y_i 相关的计算是由 Server 完成的，Client 在最终验证阶段无法直接解除自己集合元素的签名。因此，协议 1.0 暂时是安全的。

Efficiency 从协议 1.0 的第 9 步可以看出，为了保证安全性，由 Server 完成 y_i 相关的计算，需要和自己集合的元素两两组合，这使协议的通信复杂度由 $(v + w)$ 升到了 $O(vw)$ 。此外，这一版本的协议 1-5 步可以离线完成，从而提高协议的效率。

这一版本的协议可以对标 De Cristofaro et al., 2010 中 APSI Protocol derived from RSA-PPIT[de2010practical]。在他们的工作中，进一步采用了一些计算上的技巧将通信复杂度从 $O(vw)$ 降到了 $O(v + w)$ ，我们也尝试将相似的方法用在我们的协议，但是并没有成功。

4.6 Blind RSA-based APSI Protocol (Version 2.0)

Blind RSA-based APSI Protocol

- Common input: $n_1, e_1, n_2, e_2, H(), H'(), N = n_1 n_2$
- CA's input: d_1
- Client's input: $\mathcal{C} = \{hc_1, hc_2, \dots, hc_v\}$, where: $hc_i = H(c_i)$
- Server's input: $d_2, \mathcal{S} = \{hs_1, hs_2, \dots, hs_w\}$, where: $hs_j = H(s_j)$

- 1 : Client: Calculate $\forall i, x_i = hc_i R_{c:i}^{e_1} \bmod N, y_i = hc_i R_{c:i}^{e_2} \bmod N, z_i = hc_i R_{c:i}^{e_1 e_2} \bmod N$
- 2 : Client: Send $hc_1, hc_2, \dots, hc_v, R_{c:1}, R_{c:2}, \dots, R_{c:v}$ to CA
- 3 : CA: Calculate $\forall i, x'_i = x_i^{d_1} \bmod N, y'_i = y_i^{d_1} \bmod N, z'_i = z_i^{d_1} \bmod N$ then send all to Client
- 4 : Client: Send $z_1, z_2, \dots, z_v, z'_1, z'_2, \dots, z'_v, y_1, y_2, \dots, y_v, y'_1, y'_2, \dots, y'_v$ to Server
- 5 : Server: Verify whether $\forall i, z_i'^{e_1} = z_i \bmod n_1$ and $y_i'^{e_1} = y_i \bmod n_1$
- 6 : Server: If validation is ok: Calculate $\forall i, z''_i = z_i'^{d_2} \bmod N, y''_i = y_i'^{d_2} \bmod N, \forall j, t'_j = H'(hs_j^{d_2(1-e_1)} \bmod n_1)$ to Client
- 7 : Server: Send $z''_1, z''_2, \dots, z''_v, y''_1, y''_2, \dots, y''_v, t'_1, t'_2, \dots, t'_w$ to Client
- 8 : Client: Calculate $\forall i, t_i = H'((z''_i x_i'^{-1} y_i''^{-1} R_{c:i})^{e_1} hc_i \bmod n_1)$
- 9 : Client OUTPUT: $\{t_1, t_2, \dots, t_v\} \cap \{t'_1, t'_2, \dots, t'_w\}$

4.7 Analysis about our protocol 2.0

Correctness 2.0 版本协议的原理仍然是基于 0.1 版本的协议，因此其正确性可由 0.1 版本协议的正确性 4.1 保证。

Security 相比于协议 0.1 版本 4.2，这一版本的协议在第 5 步加入了 Server 对 Client 发送过来的全部数据的签名验证。由于 Client 没有 CA 的私钥，因此无法自己同时生成 hc_{fake} 和 $hc_{fake}^{d_1}$ ，故 Client 不能通过之前的方法 4.1 进行攻击。因此，协议 2.0 暂时是安全的。

Efficiency 从协议 2.0 的具体过程可以看出，通信复杂度是 $(v + w)$ 。此外，这一版本的协议中，Client 只需要和 Server、CA 各进行一轮通信即可完成集合求交的任务。

由上面的分析可知这一版本的协议有着很低的通信复杂度，并且也可以保证安全性与正确性。但是由于 Client 要发给 Server 的所有数据（不仅仅是 Client 集合中的元素）都需要 CA 进行签名，因此增加了 CA 的负担。

第 5 章 Experiments and Results

5.1 Overview

我们复现了 De Cristofaro 等人的高效 PSI 协议，并在此基础上进行改进，编写了基于证书的 APSI 协议的代码。代码仓库位于<https://github.com/Layotiver/apsi>。

我们基于了 `pycryptodome` 包的哈希，取大质数，取随机数等功能，编程实现了扩展欧几里得算法，求乘法逆元，RSA 公私钥生成，RSA 加密等算法。这部分代码位于 `myutils.py`。

本次实验中，隐私集合的元素是英语单词，隐私集合求交即求出双方共有的单词。客户端和服务端端的单词分别保存在 `client_set.txt` 和 `server_set.txt` 中。其中客户端的单词为 *text, corpus, from, language, approach, resource*，服务器端的单词为 *This, is, quite, a, departure, from, the, earlier, approach, in, NLP, applications*。其中 *from* 和 *approach* 是双方共有的单词。隐私集合求交的结果会输出这两个单词在 Client 集合里的下标，即 2 和 4。对于哈希，我们使用 `sha256` 函数，并通过设置不同的 `salt` 值来产生不同的哈希函数。

5.2 PSI protocol

我们在 `psi` 包里实现了 De Cristofaro 等人的 PSI 协议。其中 `Client` 类和 `Server` 类对于算法中的 Client 和 Server。两个类都有 `off_line` 和 `on_line` 函数，对应协议中的离线和在线处理的部分。

运行 `main_psi_server.py` 后，服务端程序启动。服务端会先进行 `off_line` 操作，然后监听本地的 12121 端口，接收来自客户端的 PSI 请求。接着运行 `main_psi_client.py`，客户端先进行 `off_line` 操作，然后将协议中的需要传输的数据发送到本地的 12121 端口。服务器收到数据后进行一系列处理，即协议中的 `on_line` 部分，再发回处理过的数据给客户端。客户端收到数据后也进行 `on_line` 部分的操作，最后算出双方共有的集合元素。

运行 `main_psi_client.py` 后，控制台打印出 2,4，代表计算出隐私集合里第 2 个和第 4 个元素为共有的元素，结果正确，参见图 5.1。

5.3 APSI protocol

De Cristofaro 等人的 PSI 协议无法验证客户端的隐私集合的真实性。比如，客户端可以谎称自己的集合里有单词 *departure*（实际上并没有），经过上述 PSI 协议，最后客户端能够知道服务器端是否有 *departure* 这个单词。因此我们需要授权的隐私集合求交（APSI）。我们实现了基于可信第三方证书的 APSI，客户端的每个元素都带有 CA 的证书。

5.3.1 Normal case

我们在 `apsi` 包里实现了基于证书的 PSI 协议，这包里同样有 `Client` 和 `Server` 类。另外，`apsi` 包里还有一个 `CertificateAuthority` 类，用于给客户端的元素颁发证书。

同样的，运行 `main_apsi_server.py` 后，服务端程序启动。服务端会先进行 `off_line` 操作，然后监听本地的 12122 端口，接收来自客户端的 APSI 请求。接着运行 `main_apsi_client.py`，客户端先进行 `off_line` 操作，这里客户端的 `off_line` 操作需要传入一个 CA 实例用于颁发证书。之后客户端按照协议将需要传输的数据发送到本地的 12122 端口。服务器收到数据后先进行证书验证。证书验证无误后，服务器端便按照协议进行数据处理，最后将处理过的数据返回给客户端。客户端收到数据后也进行 `on_line` 部分的操作，最后算出双方共有的集合元素。若证书验证错误，则客户端会收到服务器端的报错信息。

运行 `main_apsi_client.py` 后，控制台打印出 2,4，代表计算出隐私集合里第 2 个和第 4 个元素为共有的元素，结果正确，参见图 5.2。

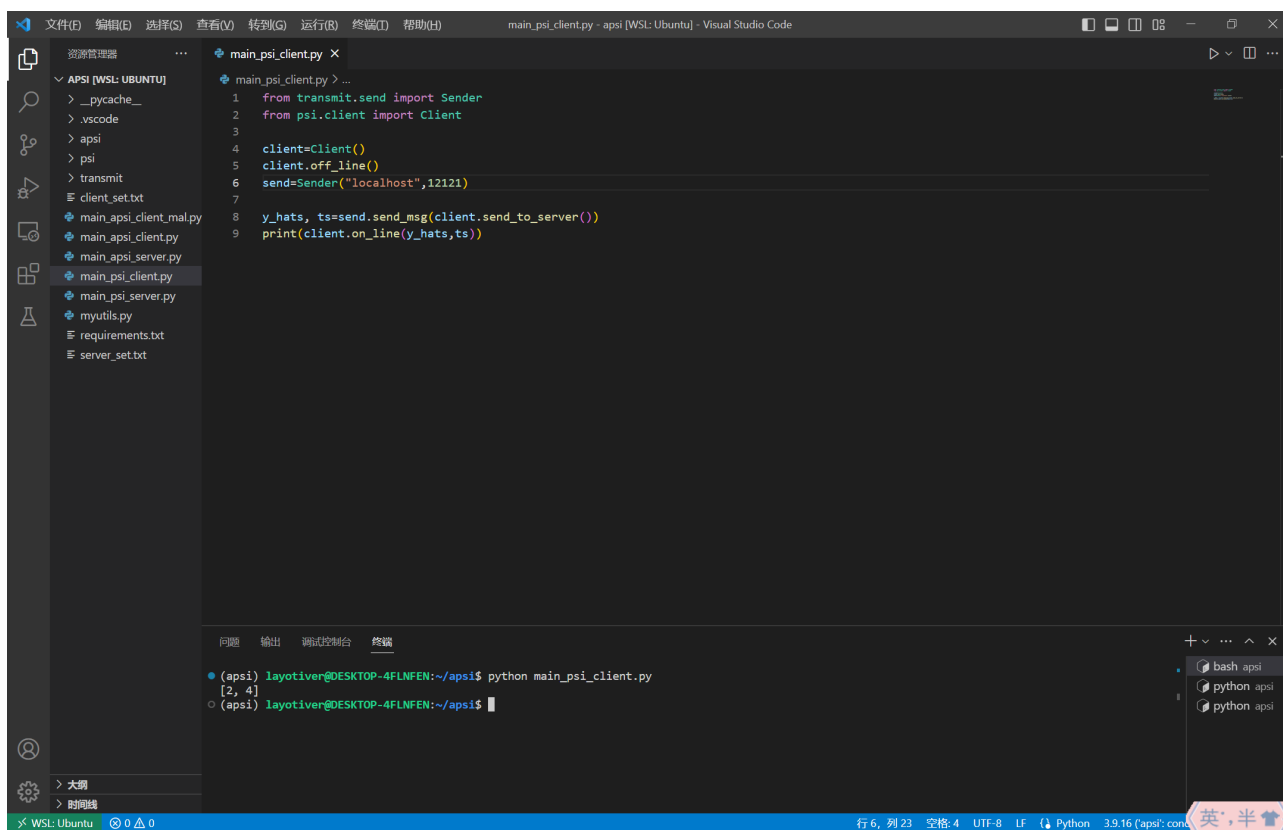


图 5.1: Blind RSA-based PSI Protocol with linear complexity

5.3.2 Malicious case

我们设计了客户端带上伪造证书的情况，测试服务器能否检测到证书造假。

运行 `main_apsi_client_mal.py`，此时客户端的 `off_line` 操作传入的 CA 实例为 `None`，客户端会生成一份全为 0 的假证书。同样将这些信息发送给服务器，并打印服务器的返回信息。控制台输出 -1，即设定了服务器报错信息，测试成功，参见图 5.3。


```

main_apsi_server.py
1 from transmit.receive import Receiver
2 from apsi.server import Server
3
4 rec=Receiver("localhost",12122)
5 server=Server()
6 server.off_line()
7
8 def func(data):
9     try:
10         ret=server.on_line(*data)
11         return ret
12     except Exception as e:
13         print(e)
14         return "SOMETHING ERROR!"
15
16 rec.rec_msg(func)

```

```

(apsi) layotiver@DESKTOP-4FLNFEN:~/apsi$ python main_apsi_client.py
[2, 4]
(apsi) layotiver@DESKTOP-4FLNFEN:~/apsi$

```

图 5.2: Blind RSA-based APSI Protocol with linear complexity: Normal

```

main_apsi_client_mal.py
1 # 恶意客户端
2 # 提供的元素没有证书
3
4 from transmit.send import Sender
5 from apsi.client import Client
6
7 client = Client()
8 client.off_line(None)
9
10 send = Sender("localhost", 12122)
11
12 print(send.send_msg(client.send_to_server()))
13

```

```

(apsi) layotiver@DESKTOP-4FLNFEN:~/apsi$ python main_apsi_client.py
[2, 4]
(apsi) layotiver@DESKTOP-4FLNFEN:~/apsi$ python main_apsi_client_mal.py
-1
(apsi) layotiver@DESKTOP-4FLNFEN:~/apsi$

```

图 5.3: Blind RSA-based APSI Protocol with linear complexity: Malicious

第 6 章 Discussion

6.1 Summary

本次调研我们先花了较多的时间精读了 5 篇文献，尽可能全面地了解当前 PSI 技术的各种实现方法，随后针对其中 De Cristofaro 等人的工作 [de2010practical] 进行了复现，以及一定程度地完善、创新。

针对 De Cristofaro et al. 提出的 Blind RSA-based PSI Protocol，我们为了防止 malicious Client 通过伪造元素的方法骗取 Server 集合中元素的信息，通过 CA 将 Client 集合中的元素加上签名，从而使 Client 无法谎称自己拥有某元素。随后我们针对初始版本协议提出了一种攻击方法，并详细分析了原因。最终我们提出了两种解决方法，并且进行了理论分析和实验验证。

我们提出的两个协议的可行性都是有理论和实验保障的，但是对于安全性我们只分析了用来攻击初始版本协议的方法是不可行的，并且暂时没有想到可行的攻击方法，并没有对安全性进行严格的理论证明。这有可能作为未来的工作。协议执行的效率方面，我们达到了和 De Cristofaro et al. 的协议相同量级的效率。

从协议的设计上看，Client 和 Server 的通信过程中需要 CA 的参与，这会增加 CA 的负载，而且协议不能算是严格意义上的两方通信。另外，在实验的过程中，由于网络的问题，我们在实际实验中并没有进行大规模集合的隐私求交，而且 Client 和 Server 是在同一台服务器的不同端口进行的通信。

6.2 Future work

未来的工作可能包括以下方向：

1. 针对我们 1.0 版本的协议，使用 De Cristofaro et al. 提出的计算上的技巧将通信复杂度将为线性。
2. 进行更为精妙的设计，实现 CA 在协议 online 阶段不参与的协议。
3. 做更充分的实验：进行更大规模的隐私集合求交，解决网络问题以提高系统的健壮性。