



隐私集合求交系统的分析与设计

Design and analysis of PSI system

学生：徐晨阳

导师：陈志立

华东师范大学软件工程学院

2022.5



目录

引言

- 选题背景
- 问题描述
- 研究现状
- 本文工作

分析设计

- 架构总览
- 核心算法
- 工程测试
- 效果展示

结语

- 工作总结
- 参考文献





选题背景

我们在注册社交账号时，通常会被询问是否要关联通讯录好友。这是一个典型的集合求交问题，具体为计算用户通讯录和平台数据库所有用户的交集。但随着隐私合规需求的日益增长，现实存在两点限制：

- 用户不愿泄漏通讯录好友的信息
- 平台无权泄露数据库用户的信息

因此需要使用隐私集合求交协议 (Private Set Intersection, PSI) 计算交集。该协议保证成功计算交集的同时，不泄露任何参与方的信息。

问题描述

设参与方为 Alice 和 Bob, 分别拥有集合 $A = \{a_1, a_2, \dots, a_n\}$, $B = \{b_1, b_2, \dots, b_m\}$, 基数 $|A| = n$, $|B| = m$, 不妨设 $m \leq n$ 。在 Alice 不知道 $\forall b_i$, 且 Bob 不知道 $\forall a_j$ 具体值的情况下, 计算集合 $C = A \cap B$, 如图 1 所示。

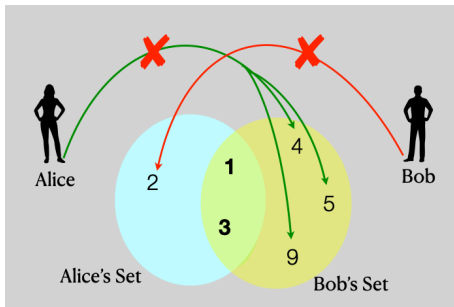


图: 问题描述



相关工作

学术界有关 PSI 协议的资源比较丰富，目前最高效的方法 [PSZ14] 采用了不经意传输协议 (Oblivious Transfer, OT)，该方法采用布谷鸟哈希 (Cuckoo Hash, CH) 来优化中间结果的集合尺寸，以减少通讯开销，同时利用少量的公钥加密完成大量不经意传输，优化了时间上的运行效率 [KKRT16]。

不经意传输协议依赖于公钥加密，可以使用 RSA 算法 [Wik22] 实现。布谷鸟哈希的论文 [PR04] 也提供了核心方法的伪代码，供开发者实现。

在开源社区，有作者进行了相关算法和协议的理论讲解，并在 GitHub 开源了基于 Python 实现的[项目链接](#)，供开发者学习参考。



本文工作

本工作将参考已有的论文，分析并设计一个支持两方隐私求交的系统，侧重于工程实现，所有模块均需要通过测试样例。

核心点在于设计相关数据结构，分析基本的密码学算法，采用不经意传输协议，利用合适的进程间通信模型，来实现理论上安全的两方隐私集合求交系统。

技术选型方面，考虑到高性能的要求，选用 Cpp 11 在 MacOS 环境下编写，利用 CMAKE 生成跨平台的 MakeFile 文件，并使用 Clang 编译。全部代码开源在 GitHub，可以根据 README 配置环境并运行，具体参考[链接](#)。



架构总览

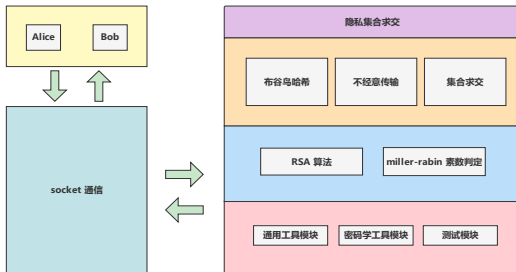
系统采用自底向上分层的架构模式，底层模块支持上层服务，通过分层进行了功能解耦。

系统开放 Alice 和 Bob 两个输入端，两方通过 Socket 通信进行数据的传输。

传输过程中，两方调用系统为其提供的接口完成 PSI 的计算。

系统架构如图 2 所示。

架构总览



图：系统架构



隐私比较

隐私比较是隐私求交的原子操作，旨在判断 $a = b$ 是否成立。首先将 a, b 拓展到 L -bit 长度，并由 Alice 生成 L 个随机数对 K_i ，形成一个随机数组 K 。

对于 b 的第 i 位 θ ，Bob 与 Alice 进行一次不经意传输，得到 $K_{i, \theta}$ ，一共计算 L 次，最终计算出 $k_b = K_{0, v_0} \oplus K_{1, v_1} \oplus \dots \oplus K_{L-1, v_{L-1}}$ 。

由于 Alice 本身知晓 K ，因此不需要进行不经意传输，只需要按照同样的规则进行 L 次计算，便可以计算出 k_a 。最终只需要比较 $k_a = k_b$ 是否成立即可，具体如图 3 所示。



隐私比较

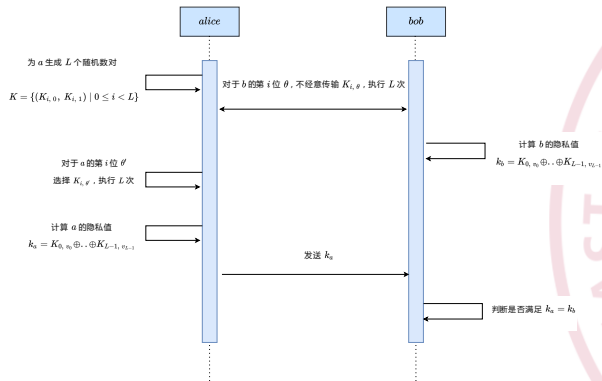


图: 隐私比较



隐私求交

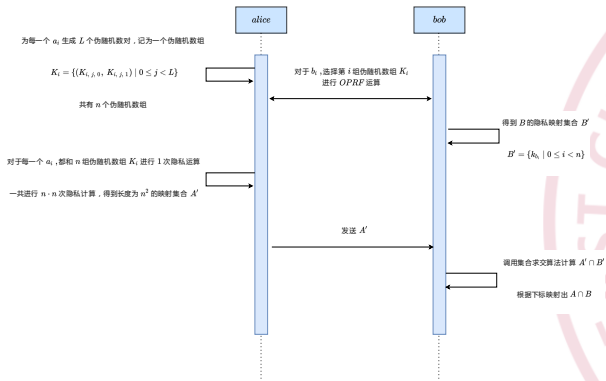
将隐私比较拓展，便可以得到隐私求交。首先 Alice 为集合 A 中的每一个元素 a_i 都生成一个随机数组 K ，一共生成 n 个。

对于集合 B 中的每一个元素 b_j ，Bob 都与第 a_j 做一次隐私比较。考虑到 $m \leq n$ ，因此一共进行 m 次隐私比较。计算完毕后，Bob 将 m 个计算结果写入集合 B' 。

对于集合 A 中的元素 a_i ，由于 Alice 不知道 a_i 与哪一个 b_j 进行了隐私比较，因此 Alice 会将 a_i 与所有 K 做隐私计算，一共进行 $n \cdot n$ 次计算，并把 n^2 个计算结果写入集合 A' 。

最后只需要计算 $C' = A' \cap B'$ ，并根据下标映射回原集合便能得到 $C = A \cap B$ ，具体如图 4 所示。

隐私求交



图：隐私求交



哈希优化

集合 A' 的基数为 $\mathcal{O}(n^2)$ ，这是通信开销的瓶颈所在。深层次的原因在于，Alice 不确定 a_i 与哪一个 b_j 进行了隐私比较，因此只能为每一个 a_i 都维护 n 个隐私值。

如果我们精确找到使得 $a_i = b_j$ 的 i ，就可以减少 a_i 做隐私计算的次数，那么集合 A' 的大小便可以得到减少。事实上，我们只需要选择一个合理的哈希表 T ，以及其对应的哈希函数 h ，满足 h 的值域是 $[0, n)$ ，问题就可以迎刃而解了。

在布谷鸟哈希的场景下，鉴于其优秀的负载能力，我们限定负载因子 α 落在 $[0.75, 1]$ 之间，那么哈希表的尺寸大小落在 $[n, \frac{4}{3} \cdot n]$ 之间，同时保证每个桶里至多只有一个元素。我们选择两个哈希函数 h_1, h_2 来更有效地处理冲突，那么集合 A' 的尺寸为 $2 \cdot n$ ，集合 B' 的尺寸为 n ，因此我们成功把集合 A' 的大小从 $\mathcal{O}(n^2)$ 降低到了 $\mathcal{O}(n)$ ，进而优化了通讯效率，具体如图 5 所示。

哈希优化

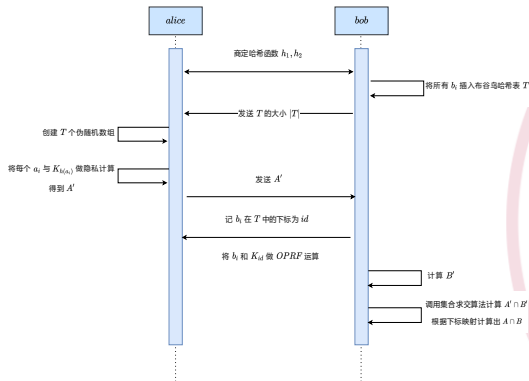


图: 哈希优化

功能测试

本系统对所有模块都进行了功能测试，确保了模块的正确性，如图 6, 7, 8, 9, 10, 11 所示。

```

$ build (master) x ./bin/crypt_test
OT TEST
case# 0:
choose_bit = 1
actual_choose_bit = 1
Correctness TEST Success!
case# 1:
choose_bit = 0
actual_choose_bit = 0
Correctness TEST Success!
case# 2:
choose_bit = 1
actual_choose_bit = 1
Correctness TEST Success!
case# 3:
choose_bit = 0
actual_choose_bit = 0
Correctness TEST Success!
case# 4:
choose_bit = 1
actual_choose_bit = 1
Correctness TEST Success!

```

图：不经意传输测试

```

$ build (master) x ./bin/blake2b_test
CASHO TEST
case# 0:
data size: 100
actual_size: 100
duration: 1.141 ms
Correctness TEST Success!
case# 1:
data size: 1000
actual_size: 1000
duration: 1.547 ms
Correctness TEST Success!
case# 2:
data size: 10000
actual_size: 10000
duration: 1.922 ms
Correctness TEST Success!
case# 3:
data size: 100000
actual_size: 100000
duration: 2.287 ms
Correctness TEST Success!
case# 4:
data size: 1000000
actual_size: 1000000
duration: 2.657 ms
Correctness TEST Success!

```

图：布谷鸟哈希表测试

```

$ build (master) x ./bin/crypt_test
PRIME GENERATION TEST
case# 0:
val = 818961331
Correctness TEST Success!
case# 1:
val = 886221221
Correctness TEST Success!
case# 2:
val = 935758541
Correctness TEST Success!
case# 3:
val = 618203297
Correctness TEST Success!
case# 4:
val = 763888213
Correctness TEST Success!

```

图：素数生成测试

```

$ build (master) x ./bin/crypt_test
PI TEST
data size = 100
actual_size: 100
duration: 1.141 ms
Correctness TEST Success!
case# 1:
data size = 1000
actual_size: 1000
duration: 1.547 ms
Correctness TEST Success!
case# 2:
data size = 10000
actual_size: 10000
duration: 1.922 ms
Correctness TEST Success!
case# 3:
data size = 100000
actual_size: 100000
duration: 2.287 ms
Correctness TEST Success!
case# 4:
data size = 1000000
actual_size: 1000000
duration: 2.657 ms
Correctness TEST Success!

```

图：隐私集合求交测试

```

$ build (master) x ./bin/si_test
SI TEST
case# 0:
data size = 100
ans1.size = 54, ans2.size = 54
Correctness TEST Success!
case# 1:
data size = 1000
ans1.size = 479, ans2.size = 479
Correctness TEST Success!
case# 2:
data size = 10000
ans1.size = 4978, ans2.size = 4978
Correctness TEST Success!
case# 3:
data size = 100000
ans1.size = 50009, ans2.size = 50009
Correctness TEST Success!
case# 4:
data size = 1000000
ans1.size = 499273, ans2.size = 499273
Correctness TEST Success!

```

图：集合求交测试

```

$ build (master) x ./bin/crypt_test
RSA TEST
case# 0:
pval = 518259, e = 1103
dval = 518259, e = 1103
C = 422318
Correctness TEST Success!
case# 1:
pval = 518259, e = 1103
dval = 518259, e = 1103
C = 422318
Correctness TEST Success!
case# 2:
pval = 518259, e = 1103
dval = 518259, e = 1103
C = 422318
Correctness TEST Success!
case# 3:
pval = 518259, e = 1103
dval = 518259, e = 1103
C = 422318
Correctness TEST Success!
case# 4:
pval = 518259, e = 1103
dval = 518259, e = 1103
C = 422318
Correctness TEST Success!

```

图：rsa 测试



性能测试

本系统还对模块进行了性能测试，通过性能测试验证理论分析和设计的正确性。

首先验证布谷鸟哈希的性能。我们执行多次随机的插入、删除、查询操作，将运行时间和负载因子大小和与标准库里的 `std::unordered_map` 对比，如表 1 和 2 所示。

由数据发现，在小数据情况下，前者比后者略慢；在大数据情况下，前者比后者略快；不管数据量大小，前者的负载因子都更大。因此布谷鸟哈希表可以在保证负载量的情况下，支持稳定的高速查询，符合我们理论的猜想。



性能测试

表: cuckoo 和 `std::unordered_map` 的运行效率对比表

数据量级/次数	cuckoo 运行时间/ μ s	<code>std::unordered_map</code> 运行时间/ μ s
100	131	97
1,000	523	445
10,000	3,540	3,768
100,000	29,136	32,557
1,000,000	237,514	281,908
10,000,000	2,667,617	2,883,467



性能测试

表: cuckoo 和 `std::unordered_map` 的负载因子对比表

数据量级/次数	cuckoo 负载因子	<code>std::unordered_map</code> 负载因子
100	0.832031	0.556701
1,000	0.693359	0.507614
10,000	0.945312	0.507614
100,000	0.693359	0.507614
1,000,000	0.96875	0.507614
10,000,000	0.693359	0.507614



性能测试

再验证一下朴素隐私集合求交和布谷鸟哈希优化后的隐私集合求交在集合大小和运行速度上的差别，时间单位为 μs ¹，如表 3 和 4 所示。

根据实验数据发现，隐私集合 A' 的尺寸从 $O(n^2)$ 降低到了 $O(n)$ ，符合理论上的计算。当集合尺寸降低之后，对应的计算效率也提高了。

¹时间单位换算：1 s = 1,000,000 μs



性能测试

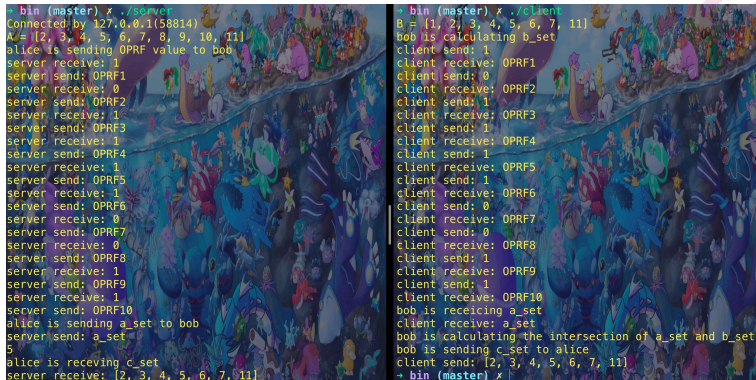
表: 朴素协议: 隐私集合指标和数据量的关系表

数据量级	集合大小	运行时间/ μs
100	10,000	14,669
1,000	1,000,000	1,031,746

表: 优化协议: 隐私集合指标和数据量的关系表

数据量级	集合大小	运行时间/ μs
100	200	12,489
1,000	2,000	87,509
10,000	20,000	704,075
100,000	200,000	11,036,512

效果展示



```
+ bin(master) x ./server
Connected by 127.0.0.1(58814)
A=[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
alice is sending OPRF value to bob
server receive: 1
server send: OPRF1
server receive: 0
server send: OPRF2
server receive: 1
server send: OPRF3
server receive: 1
server send: OPRF4
server receive: 1
server send: OPRF5
server receive: 1
server send: OPRF6
server receive: 0
server send: OPRF7
server receive: 0
server send: OPRF8
server receive: 1
server send: OPRF9
server receive: 1
server send: OPRF10
alice is sending a_set to bob
server send: a_set
5
alice is receiving c_set
server receive: [2, 3, 4, 5, 6, 7, 11]

+ bin(master) x ./client
B=[1, 2, 3, 4, 5, 6, 7, 11]
bob is calculating b_set
client send: 1
client receive: OPRF1
client send: 0
client receive: OPRF2
client send: 1
client receive: OPRF3
client send: 1
client receive: OPRF4
client send: 1
client receive: OPRF5
client send: 1
client receive: OPRF6
client send: 0
client receive: OPRF7
client send: 0
client receive: OPRF8
client send: 1
client receive: OPRF9
client send: 1
client receive: OPRF10
bob is receiving a_set
client receive: a_set
bob is calculating the intersection of a_set and b_set
bob is sending c_set to alice
client send: [2, 3, 4, 5, 6, 7, 11]
+ bin(master) x
```

图: 效果展示



工作总结

本工作基于已有的论文，实现了一个支持两方求交的轻量级的 PSI 系统。核心工作在于实现一个基于 OT 的 PSI 协议，并在此基础上，实现由 Cuckoo Hash 优化的协议，同时兼顾工程架构、设计和测试，打磨系统的稳定性。就工作量而言，最终版本的 Cpp 代码行数为 1343 行，并通过了所有的功能测试。工作完全独创，全文 14756 字，去除引用后知网查重率为 0.00%。

但就 PSI 协议而言，可以利用扩展 OT 来进一步优化计算效率，但是这个算法实现起来有一定难度，暂时没有实现。同时为了效率，系统暂时没有支持大整数运算，如果需要支持大整数运算且保证效率，可以考虑使用例如 gmp 的第三方库。同时，本系统只是隐私计算框架的基础系统，只支持两方求交，若要支持多方计算，需要引入新的协议和算法，来支持多方求交。



参考文献

- [KKRT16] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu, *Efficient batched oblivious prf with applications to private set intersection*, Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 818–829.
- [PR04] Rasmus Pagh and Flemming Friche Rodler, *Cuckoo hashing*, Journal of Algorithms **51** (2004), no. 2, 122–144.
- [PSZ14] Benny Pinkas, Thomas Schneider, and Michael Zohner, *Faster private set intersection based on ot extension*, 23rd USENIX Security Symposium (USENIX Security 14), 2014, pp. 797–812.
- [Wik22] Wikipedia contributors, *Oblivious transfer — Wikipedia, the free encyclopedia*, https://en.wikipedia.org/w/index.php?title=Oblivious_transfer&oldid=1077709366, 2022.



结语

感谢聆听！

