基于“隐语”的恶意安全性机器学习隐私保护协议的研究与实现

摘　要

本毕设项目重点关注在“隐语”框架 SPU 密态设备中实现一种全过程支持恶意安全性的隐私保护混合协议。“隐语”框架中已经实现了三种安全多方计算协议（ABY3，Semi2K，Cheetah），然而这三种协议目前均尚未实现全过程支持恶意安全性，在实际使用场景下，如果各个参与方之间的身份无法确认，某个参与方可能会实施偏离协议的任意攻击，那么半诚实安全性就不能够继续保障安全协议的隐私性和正确性。基于此，本项目在“隐语”框架 SPU 密态设备中实现一套恶意安全性隐私保护混合协议，并测试它在训练机器学习模型场景下的性能。

这个项目中，设计并完善了一种三方诚实大多数隐私保护协议，能够满足恶意安全性，并实现了“隐语”框架 SPU 密态设备中规定的所有接口。为了实现这种协议，本项目采用了目前不使用零知识证明技术的最先进三方协议（Chida 等人，CRYPTO 2018）作为算术运算协议，使用目前先进的二进制电路协议（Furukawa 等人，EUROCRYPTO 2017）作为布尔运算协议，使用先进的基于 EdaBit 的方案（Escudero 等人，CRYPTO 2020）作为秘密共享转化协议。同时，本项目还实现了这些协议所需的常见基础设施。

本项目还测试并对比了两种离线预处理方案的性能和不同设定下训练机器学习模型的效率。通过对比的对象是目前最先进、性能最好的隐私计算框架（Marcel Keller, ACM SIGSAC 2020），本项目所实现的离线协议在性能上与他们相近，其中生成 128M 个 beaver 三元组的效率相比于 MP-SPDZ 要快 34%。在机器学习模型训练上，对比目前最好的半诚实安全性混合协议（Payman 等人，ACM SIGSAC 2018），不同参数条件下本项目实现的性能符合理论估计，其中本项目实现的恶意安全性协议与半诚实安全性协议的计算性能差距稳定在 5 倍，而精确度与明文计算几乎一致。

关键词　安全多方计算　恶意安全性　三方诚实大多数　隐语　机器学习隐私保护

Research and implementation of malicious privacy-preserve machine learning (PPML) protocols in secretflow

ABSTRACT

In this graduation project, we focus on implementing a privacy-preserving mixed protocol that achieves malicious security in SPU (Secure Processing Unit) of ”SecretFlow” framework. ”SecretFlow” has already implements three privacy-preserving protocols. However, they didn't supports malicious security throughout the process. In practical scenarios, if the identities of the parties cannot be confirmed, malicious adversaries may implement arbitrary attacks deviating from the protocol. As a result, semi-honest security cannot guarantee the privacy and correctness of the security protocol. Based on this, we aim to implement a malicious security privacy-preserving mixed protocol in SPU of ”SecretFlow” and test its performance in machine learning model training scenarios.

In this work, we implemented a three-party honest majority privacy-preserving protocol and made all the interfaces specified in SPU of ”SecretFlow” maliciously secure. To achieve this protocol, we used the state-of-the-art three-party protocol without distributed zero-knowledge proof technology (Chida et al. CRYPTO 2018) as the arithmetic operation protocol, the well-known binary circuit protocol (Furukawa et al. EUROCRYPTO 2017) as the Boolean operation protocol, and the advanced EdaBit-based scheme (Escudero et al. CRYPTO 2020) as the conversion protocol. At the same time, we also implemented the common infrastructure required by these protocols.

We also tested the performance of two preprocessing schemes, and the efficiency in training machine learning models in different settings. Our baseline is the most advanced and best-performing privacy computing framework (Marcel Keller, ACM SIGSAC 2020). Through comparison, the offline protocol we implemented is comparable in performance to theirs, with different advantages and disadvantages under different settings. For example, there is a 34% percent promotion when generating 128 million binary triples. In machine learning model training, we compared it vertically with the best semi-honest security hybrid protocol (Payman et al. ACM SIGSAC 2018). Under different parameter conditions, our performance meets theoretical estimates, which has a factor of 5 gap compared to semi-honest protocol. And the accuracy is almost same compared to plaintext computation.

KEY WORDS SMPC malicious security 3PC honest-majority SecretFlow PPML

目　录

第一章　简介

1.1　概念

安全多方计算（Secure Multi-Party Computation，简称 SMPC）是一种隐私保护技术，其核心目标是在多个参与方之间进行安全、高效的协同计算，同时确保各方的数据隐私得到保护。这一技术允许多个计算参与者合作处理敏感信息，而无需暴露各自的原始数据。在安全多方计算过程中，每个参与方将其数据加密，然后进行分发。通过对这些加密数据的计算，可以得到最终结果。在整个过程中，不会有任何一方获取到其他参与方的明文数据。这意味着每个参与方都可以确保其数据安全，同时又能与其他参与方共同完成计算任务。

安全多方计算为各种计算场景提供了隐私保护，其中之一便是机器学习领域。一种场景是，Alice 拥有机器学习模型，Bob 想要使用模型，但是 Alice 不愿意把模型交给 Bob，Bob 也不想直接把明文的数据交给 Alice 计算。为了保证数据隐私，实现这些数据的价值挖掘，机器学习隐私保护（Privacy-Preserving Machine Learning，简称 PPML）应运而生。PPML 是一种允许在不泄露原始数据的情况下进行机器学习的技术。通过结合安全多方计算、差分隐私等加密技术，实现在分布式环境中对加密数据的处理和分析。这样，各个参与方可以在保护数据隐私的同时，共享机器学习模型的构建和训练过程。

隐语（SecretFlow）[1] 是由蚂蚁集团开发的隐私保护数据分析与机器学习框架，其目标是使得数据科学家和机器学习开发者可以非常容易地使用隐私计算技术进行数据分析和机器学习建模，而无需了解底层技术细节。为达到这个目标，隐语提供了一层设备抽象，将多方安全计算 (MPC)、同态加密 (HE) 和可信执行环境 (TEE) 等隐私计算技术抽象为密文设备，将单方计算抽象为明文设备。基于这层抽象，数据分析和机器学习工作流可以表示为一张计算图，其中节点表示某个设备上的计算，边表示设备之间的数据流动，不同类型设备之间的数据流动会自动进行协议转换。在这一点上，隐语借鉴了主流的深度学习框架，后者将神经网络表示为一张由设备上的算子和设备间的张量流动构成的计算图。

隐语框架围绕开放这一核心思想，提供了不同层次的设计抽象，希望为不同类型的开发者都提供良好的开发体验。在设备层，隐语提供了良好的设备接口和协议接口，支持更多的设备和协议插拔式的接入，希望与密码学、可信硬件、硬件加速等领域专家通力合作，不断扩展密态计算的类型和功能，不断提升协议的安全性和计算性能。同时，隐语提供了良好的设备接口，第三方隐私计算协议可作为设备插拔式接入。在算法层，为机器学习提供了灵活的编程接口，算法开发者可以很容易定义自己的算法。

1.1.1　安全模型

根据攻击者的能力，可以把 MPC 分为两种模型：

- 半诚实安全性模型（Semi-Honest）：半诚实攻击者（又叫被动攻击者）诚实的遵守

协议执行，但是会尝试从协议过程中记录交互数据并学习更多超出协议规定的知识。

- 恶意安全性模型（Malicious）：恶意攻击者（又叫主动攻击者）会偏离协议执行任意策略攻击。

令恶意攻击者最多能够腐败的参与方数量为 t，根据攻击者与所有参与方数量 n 的关系，又可以把 MPC 分为三种模型：

- 不诚实大多数（Dishonest Majority）：$t = n - 1$。这对应 $n - 1$ 参与方联合来尝试获取一个诚实参与方数据的情景。
- 诚实大多数（Honest Majority）：$t < \frac{n}{2}$。很多重要的函数只可能在这种设定下达到信息论安全性。
- $\frac{1}{3}$ 腐败（$\frac{1}{3}$ Corruption）：$t < \frac{n}{3}$。最高的安全属性只能在这种设定下达到，拜占庭将军问题（Byzantine Generals Problem）指出了这一点。

其中，一方被攻陷的三方计算（即最小范围的诚实大多数场景）具有重要的研究意义，在这种场景下复制秘密共享方案[2]（Replicated Secret Sharing）效率很高，并且能够设计处更有效的协议。

1.1.2 安全定义

为了证明协议的安全性，需要一个精确的安全多方计算的定义。目前安全证明是基于模拟的方法，为了证明一个 SMPC 方案是安全的，研究者通常会构建两个模型——理想世界和现实世界。在理想世界中，参与方将输入发送给一个可信的第三方，该第三方负责计算函数并返回结果。在现实世界中，参与方通过 SMPC 协议进行计算。安全性证明的目标是表明，在现实世界中，攻击者获得的信息与理想世界中的信息等价。具体证明过程中，需要构造一个称为模拟器（simulator）的实体来证明安全性，它尝试模拟现实世界中攻击者的行为，同时只能访问理想世界中的信息。如果模拟器能够成功地生成与现实世界中攻击者观察到的输出相同的输出，那么就可以认为 SMPC 协议是安全的，因为在这种情况下，攻击者无法根据输出获得关于输入数据的额外信息。直观的看，恶意安全性模型似乎涵盖了半诚实安全性模型，换句话说，恶意安全性是严格高于半诚实安全性的。但是在这种安全定义下，理想世界的恶意参与方拥有了更多的能力，因此存在满足恶意安全性但不满足半诚实安全性的模型[3]。

下面描述其中最核心的属性：

- 隐私性：任何一方都不应该获得除其规定输出之外的信息。特别地，对于其他参与方的输入，唯一可以了解的是从输出本身可以推导出的信息。
- 正确性：保证每个参与方收到的输出都是正确的。

- 输入独立性：腐败方必须独立于诚实方的输入选择自己的输入。这个属性在秘密拍卖中非常重要，因为出价被保密，各方必须独立地确定出价。需要注意的是，输入独立性并不意味着隐私性。
- 输出交付保证：腐败方不能阻止诚实方收到他们的输出。换句话说，对手不能够通过进行类似“拒绝服务”攻击来干扰计算。
- 公平性：只有当诚实方收到他们的输出时，腐败方才能收到他们的输出，而不允许出现腐败方获得输出而诚实方没有获得输出的情况。这个属性在合同签署的情况下非常关键。具体来说，如果受损方获得了签署的合同而诚实方没有获得，这对诚实方来说是不公平的。需要注意的是，输出交付保证意味着公平性，但反之则不一定成立。

需要注意到，有时候安全计算的定义会放宽条件，不包括公平性和输出交付保证。这种放宽条件的安全级别被称为“带中止的安全”（Security with Abort），这种结果是，攻击者可能会获得输出，而诚实方则不能。这种放宽的原因主要有两点。首先，在某些情况下，实现公平性是不可能的（例如，两个方之间的公平硬币抛掷是不可能的）。其次，在某些情况下，如果不保证公平性，已知的协议会更加有效。因此，如果应用程序不需要公平性（特别是在只有一个方需要输出的情况下），这种妥协是有用的。

1.2 国内外研究现状

MP-SPDZ MP-SPDZ 框架[4] 基于 SPDZ-2 开发，实现了大量的安全协议，并且提供三种不同类型的秘密共享技术（Replicated Secret Sharing, Shamir Secret Sharing, Yao’s Garbled Circuit Sharing）和多种安全模型。在实现上，该框架基于 C++ 开发，将 MPC 原语设计为虚拟机来提供计算接口；上层应用使用 Python 语言，并通过 Python 编译器和优化将上层语言转化为字节码，交由底层虚拟机执行，以此来确保执行效率。同时该框架支持 tensorflow 前端和部分 pytorch 前端，以兼容主流机器学习模型。该框架拥有很高的执行效率，例如，使用姚氏混淆电路秘密共享方案计算 64-bit 整数的向量内积，其性能要比 ObliVM[5] 快 77 倍[4]。得益于该框架的性能，MP-SPDZ 一度成为学术界最常用的运行基准测试的框架。

然而，随着 MP-SPDZ 框架的更新和迭代，其底层代码变得越来越复杂，代码可读性、可维护性差。并且，由于该框架的编译部分处理比较简单，因此对于使用框架的人员来说，编写代码的限制较多，同样的在编译层面的效率也较低。

CrypTen CrypTen 框架[6] 由 Facebook 研究团队开发，在 python 层面提供 MPC 的支持，API 的设计优先考虑机器学习，并且其 API 设计密切遵循 PyTorch 框架的规则，同时 CrypTen 提供 GPU 运算，因此在机器学习方面提供了很高的效率。框架提供了半诚实安全性协议与自动求导等，不过，CrypTen 更像是 PyTorch 框架的扩展，专注于机器学习，因此在可扩展性方面并不优秀。

ABY3 ABY^3 框架[7] 建立在三方的诚实大多数模型，并且提供了恶意安全性协议，论文中提出了高效的 Arithmetic-Binary-Yao 秘密共享转化方案，并且提供了浮点数截位、向量乘法、特殊混合运算的高效协议。得益于三方诚实大多数的设定，相同模型下 ABY^3 的 64-bit 整数向量内积效率比 MP-SPDZ 框架仍快 50%[4]。

1.3 本文主要工作

本项目为蚂蚁“隐语”框架提供了恶意安全性混合协议的支持，并能够完成机器学习层面的隐私训练与计算，这是隐语当中首个全过程支持恶意安全性的混合协议[1]，为业界提供了新的可选项。除此之外，本项目为隐语下 SPU 框架提供了大量基础设施，本项目的顺利进行也证明了 SPU 框架执行复杂任务的可行性，为后续学术界进行科研、实验提供了基础与更多选择。

1-out-of-3 恶意安全性的混合协议 在“隐语”框架实现了高效的三方诚实大多数恶意安全性协议。恶意攻击者拥有了比半诚实攻击者更强大的能力：恶意攻击者可以偏离协议执行任意的攻击，显然对于任何半诚实安全性协议来说，恶意攻击者都能够破坏数据的完整性并且不会被任何诚实方感知，这在实际场景，尤其是在参与方身份不能确保诚实的场景下是非常危险的。本项目所实现的恶意安全性协议可以达到 Security with Abort 条件的安全性，即能够确保数据的正确性和隐私性。本项目的实现提供了完备的接口，可以让上层应用轻松地利用该混合协议进行安全多方计算。此外，本项目还针对性地优化了通信轮数，以提高系统性能。

协议可以分为以下三部分：

- 在算术运算部分使用 Spdzwise Field 协议[9]，这是一种基于 SPDZ 秘密共享方案[10] 的高效协议，可以用于检查数据的完整性以抵抗恶意攻击者的攻击。实现中，使用 61-bit 梅森素数域以提升域运算的效率。

- 在布尔运算部分使用 FLNW17 协议[11]，这是不使用分布式零知识证明技术下一种先进的二进制协议[2]。它基于复制秘密共享方案，使用 Beaver 乘法三元组方法[13] 和切片选择（Cut-and-Choose）范式[14] 来生成 beaver 三元组。在三方场景下，每个参与方在合适的参数设定下，计算一个与门需要 10 bit 的通信量开销。同时，本文将 SPU 中实现的 Cut-and-Choose 范式，同 MP-SPDZ 框架中 Cut-and-Choose 范式进行了性能和通信上的对比。

- 使用基于 EdaBit[15]（Extended daBit）的方法来进行数据在算术部分和布尔部分的转化，其中 EdaBit 是 daBit[16]（double-shared authenticated Bit）的扩展形式。这种

[1] “隐语”框架拥有 SPD$\mathbb{Z}_{2^k}$[8] 协议，但是该协议的离线（offline）阶段尚未完成恶意安全性，因此整体来看该协议并没有做到全过程支持恶意安全性。

[2] 目前诚实大多数恶意安全性三方协议中，Araki 等人的工作[12] 是不基于分布式零知识证明的最优方案，其每个 AND 门计算的通信开销是 7bits，但是由于前期调研不充分，导致在本工作没有使用这种方法。

技术同 FLNW17 方案相似，需要在前置阶段生成 EdaBits 用于做数据转化，同时需要切片选择技术来确保离线过程的恶意安全性。同样的，本文将 SPU 中实现的生成 EdaBit，同 MP-SPDZ 框架中生成 EdaBit 进行了性能上的对比。

恶意安全性的机器学习模型应用 本项目将这种恶意安全性混合协议应用到训练机器学习模型中。本项目测试了明文计算、ABY^3 协议[7]、上述混合协议在不同参数条件下，训练线性回归机器学习模型的效率，并针对数据进行了分析。其中本文实现的恶意安全性协议在不同参数下大约均比半诚实协议慢 5 倍，而这种常数性能差距实际上是完全能够被接受的。

第二章　基础知识

2.1　符号说明

令 $\mathcal{P} = \{P_0, P_1, P_2\}$ 表示三个参与方的集合，本文使用 $P_i, i \in \{0,1,2\}$ 来表示集合中的一个特定的参与方，并且使用 $P_{i\pm1}$ 表示下一个（+）或者上一个 − 参与方，这里的 + 和 − 为循环运算。

本文使用 $\mathbb{F}_2$ 来表示一个二进制域，即 $\{0,1\}$；使用 $\mathbb{F}_p$ 来表示一个模 p 运算的有限域，其中 p 的比特长度为 m，即 $m = \log p$。

令 κ 表示安全参数，本文使用 $\mathcal{F} = \{F_K | K \in 0,1^\kappa, F_K : \{0,1\}^\kappa \to R\}$ 来表示一个伪随机函数家族，其中提供密钥 $K \in \{0,1\}^\kappa$，伪随机函数 F_K 接收一个输入 $x \in 0,1^\kappa$ 并输出一个伪随机序列 $y \in R$，其中 R 表示 $\mathbb{F}_2$ 或者 $\mathbb{F}_p$。

2.2　秘密共享方案

秘密共享（Secret Sharing）是一种密码学技术，旨在将一个秘密信息分割成多个份额，这些份额可以分发给一组（#n）参与者。这种方法的目的是确保只有在获得一定数量（#t）的份额时，才能重新构建原始秘密信息。这样，即使某些参与者被攻击者入侵，秘密信息仍能保持安全可用。

2.2.1　加法秘密共享方案

加法秘密共享方案（Additive Secret Sharing）是一种常用的秘密共享技术，在加法秘密共享方案中，秘密值的重建基于环或域上的加法运算。例如，对于一个 $\mathbb{F}_p$ 上的 n 方加法秘密共享方案，秘密数据 $x \in \mathbb{F}_p$ 会被分解为 n 个随机数 $x_0, x_1, \ldots, x_{n-1} \in \mathbb{F}_p$，使得 $x = \sum_{i=0}^{n-1} x_i$，然后参与方 P_i 将会获得份额 x_i。显然，这种方案下只有当所有参与方都提供自己的秘密份额 x_i 并求和，才能够完成秘密恢复。因此，这种方案的门限 $t = n$，并且能够抵抗拥有 $n-1$ 个恶意参与方（不诚实大多数）情况下的攻击。

2.2.2　复制秘密共享方案

复制秘密共享方案（Replicated Secret Sharing, RSS）[2] 是对加法秘密共享方案的扩充。在这种方案下，秘密数据 $x \in \mathbb{F}_p$ 仍然会按照加法秘密共享方案的形式进行分解，然而，每个参与方会得到其中的若干个数据而不是一个。以三方复制秘密共享方案为例，秘密数据 $x \in \mathbb{F}_p$ 被分解为三个随机数 x_0, x_1, x_2，使得 $x_0 + x_1 + x_2 = x$。然后，对于参与方 P_i，会得到份额二元组 (x_i, x_{i+1}) 来当作他的一份份额。可以看出，在这种秘密份额分发的方式下，任何两个参与方都拥有足够恢复出秘密数据的份额，因此这种三方复制秘密共享方案所具有的门限值为 2。对于 x 在 $\mathbb{F}_p$ 上复制秘密共享，本文使用 $[\![x]\!]^F$ 来表示，即 $[\![x]\!]^F = (x_0, x_1, x_2)$，而对于其在 $\mathbb{F}_2$ 上的复制秘密共享，本文使用 $[\![x]\!]^B$ 来表示。

2.2.3 SPDZ 可认证秘密共享方案

SPDZ 可认证秘密共享方案可以说是一个更上层的思想，其核心思想是设置一个全局的消息认证码（MAC）密钥 Δ，并通过消息认证码 MAC 来确保秘密共享不会被恶意攻击者篡改。具体来讲，这个全局 MAC 密钥以秘密共享的形式（$[\![\Delta]\!]$）分布在各个参与方下，在协议执行全过程中均不会被打开。对于每一个输入 $[\![x_i]\!]$，参与方共同计算出 $[\![t_i]\!] = [\![\Delta]\!] \cdot [\![x_i]\!]$，并将 $[\![t_i]\!]$ 与 $[\![x_i]\!]$ 共同保存。在这种情况下，攻击者若想改变 $[\![x_i]\!]$ 对应于自己的 share，其必须还要改变对应的消息认证码。但是攻击者对全局消息认证码密钥的信息一无所知，因此攻击者不存在高效的方法伪造消息认证码。

2.2.4 秘密共享方案的同态性

加法同态性 通过上面的秘密共享方案可以看出，两种秘密共享方案都具有加法同态性。例如，加法秘密共享方案中，假设两个秘密数 $x, y \in \mathbb{F}_p$，对应的秘密共享分别为 $(x_0, x_1, x_2), (y_0, y_1, y_2)$，显然，如果直接对两个秘密共享进行相加，即得到：$[\![x]\!] + [\![y]\!] = (x_0 + y_0, x_1 + y_1, x_2 + y_2)$，可以看出，此时的秘密共享恢复出的数据就是 $x + y$ 的值，即 $(x_0 + y_0) + (x_1 + y_1) + (x_2 + y_2) = (x_0 + x_1 + x_2) + (y_0 + y_1 + y_2) = x + y$，也就是说，如果用秘密共享的符号表示的话即为 $[\![x]\!] + [\![y]\!] = [\![x + y]\!]$。同理，对于复制秘密共享方案和 SPDZ 可认证秘密共享方案来说也具备此性质。

乘法同态性 此外，对于一个公开的常数乘以某个秘密共享的情况也具备同态性。设公开常数为 $c \in \mathbb{F}_p$，秘密数 $x \in \mathbb{F}_p$ 的秘密共享为 $[\![x]\!]$，那么，如果对 $[\![x]\!]$ 的每个份额都乘以公开常数 c，即得到新共享值为 (cx_0, cx_1, cx_2)，可以看出，此时的秘密共享恢复出的数据就是 cx 的值，即 $cx_0 + cx_1 + cx_2 = c(x_0 + x_1 + x_2) = cx$。用秘密共享表示的话即为 $c[\![x]\!] = [\![cx]\!]$。这说明了上述三种秘密共享方案满足对常数的乘法同态性。

关于常数与秘密数加法的同态性 这里另外要提到的一点是，对于一个秘密数和一个公开常数的加法运算也可以称为具有同态性，这取决于如何定义一个常数和一个秘密共享的加法。例如，设公开常数为 $c \in \mathbb{F}_p$，秘密数 $x \in \mathbb{F}_p$ 的秘密共享为 $[\![x]\!]$。在加法秘密共享中，如果想要得到 $x + c = x_1 + x_2 + x_3 + c$ 的值，其实只需要让某个参与方将自己的份额增加 c，而其他参与方不变，这种方式定义出的常数和秘密共享加法同样满足加法同态性。在 SPDZ 可认证秘密共享方案中，对原始数据 $[\![x]\!]$ 的加法定义很简单，而对于消息认证码部分的加法，需要在原本 MAC 基础上增加 $\Delta * c$，不过因为各参与方拥有 $[\![\Delta]\!]$，因此这个问题就转化成了一步常数与秘密共享值的乘法问题，和一步两个秘密共享值加法的问题，上文已经提到，这两步均具有同态性，因此，SPDZ 可认证秘密共享方案的常数与秘密数加法仍然具有同态性。

2.3 基于 RSS 的子协议

2.3.1 RSS 的半诚实乘法协议

在三方诚实大多数场景下，复制秘密共享方案有着很高效的秘密数乘法协议。假设有秘密数 $x, y \in \mathbb{F}_p$，他们的秘密共享分别为 $[\![x]\!] = (x_0, x_1, x_2), [\![y]\!] = (y_0, y_1, y_2)$，想要计算他们的乘法，其实就是计算 $x \cdot y = (x_0 + x_1 + x_2) \cdot (y_0 + y_1 + y_2) = \sum_{i=0}^{2} \sum_{j=0}^{2} x_i y_j$ 的值。显然，对于任意的单项式 $x_i y_j, i, j \in \{0, 1, 2\}$，都存在一个参与方 $P_{\lambda \in \mathcal{P}}$ 能够本地计算出。换句话说，乘法展开式中的九个单项式，刚好能够由每个参与方计算其中三项得到，其中 P_i 负责计算 $z_i := x_i y_i + x_i y_{i+1} + x_{i+1} y_i$，这样 (z_0, z_1, z_2) 就构成了 $z := x \cdot y$ 的加法秘密共享。然而，由于想要构造 z 的复制秘密共享，因此还需要参与方 P_i 将计算得到的 z_i 发送给下一方，并接收上一方发来的数据，这样才能构造成 z 的复制秘密共享，用 $\mathcal{F}_{mult}$ 来表示乘法操作。

然而这种做法存在风险，因为如果参与方 P_i 直接将 $z_i = x_i y_i + x_i y_{i+1} + x_{i+1} y_i$ 发送给 P_{i+1}，参与方可以根据自身持有的秘密份额 x_{i+1}, y_{i+1} 来推算出 x_i 和 y_i 的相关性信息。如果能够将需要发送的信息先进行一次“掩码”操作再发送，那么就能够解决这个问题。这里 Araki 等人[17] 提出了一种零秘密分享（Zero share），即满足 $\sigma_0 + \sigma_1 + \sigma_2 = 0$ 的秘密分享，如果能够让三方分别得到零秘密分享的一个份额，就可以用它来掩盖原本 z_i 的值，这样进行交换后，秘密数仍然保持不变，即 $z = z_0 + z_1 + z_2 + \sigma_0 + \sigma_1 + \sigma_2$。生成零秘密分享不需要通信代价（如果不考虑协议初始化），只需要让每两个参与方之间都共享一个伪随机数生成器，即 P_i 和 P_j 共同拥有 $PRNG_{i,j} \in \mathcal{F}, i < j$，那么可以通过下面步骤来生成零秘密分享:

- 第一步，每个参与方 P_i 通过伪随机数生成器 $PRNG_{i-1,i}, PRNG_{i,i+1}$ 分别获得两个元素 $\rho_i, \rho_{i+1} \in \mathbb{F}_p$，这里 P_i 与 P_{i+1} 共同拥有 ρ_i
- 第二步，每个参与方计算 $\sigma_i = \rho_i - \rho_{i+1}$

此时，$(\sigma_0, \sigma_1, \sigma_2)$ 就构成了一个零秘密分享。

2.3.2 RSS 的构造、秘密重构与公开

构造 RSS，即将一个秘密数 x 转化为一个秘密共享。这里其实分成了三种情况：秘密数只被某个参与方 P_i 拥有，秘密数被两个参与方共同拥有，以及秘密数被所有参与方拥有。本文用 $\mathcal{F}_{input}$ 表示秘密构造。

秘密数只被某个参与方 P_i 拥有 这种情况下，ABY^3[7] 提供的方案是：首先，P_i 与 P_{i-1} 通过 $PRNG_{i-1,i}$ 获取随机数 r，P_{i-1} 将自己的秘密份额设置为 $(0, r)$，随后 P_i 计算 $x - r$ 的值并将自身的秘密份额设置为 $(r, x - r)$，最后，P_i 把 $x - r$ 的值发送给 P_{i+1}，后者将自己的秘密份额设置为 $(x - r, 0)$。显然，上述的秘密共享 $(r, x - r, 0)$ 显然是 x 的一个秘密共享，并且在传递过程中由于随机数 r 的保护，P_{i+1} 对秘密值的信息一无所知。

秘密数被多个参与方共同拥有 这种情况十分简单。假设拥有秘密数 x 的参与方为 P_0, P_1，那么只需设置 x 的秘密共享为 $(0, x, 0)$，即可满足所有要求。对于三方都知道秘密数的情况，可以设定一个共识，把它当作两方拥有秘密数的情况去做，这样也能够完成构造。

秘密重构是指：将秘密数据，通过秘密共享，重构给某一个参与方 P_i，而公开则是将秘密数据公布给所有参与方。这里仍然只探讨诚实大多数的三方场景。

在半诚实安全性下，想要重构秘密数据 x 给 P_i，只需要让 P_{i+1} 将自己的秘密份额 x_{i+2} 发送给 P_i 即可，这时 P_i 接收到 x_{i+2} 就可以计算 $x := x_0 + x_1 + x_2$。对于公开秘密数 x，只需要让每个参与方 P_i 将秘密份额 x_{i+1} 传递给 P_{i-1}，并从 P_{i+1} 接收秘密份额 x_{i+2}。这样就做到了信息公开。每个元素的重构，需要发送方发送一个域上元素（$\mathbb{F}_p$ 或 $\mathbb{F}_2$）大小的信息，而每次公开需要每个参与方都发送一个元素的信息。

在恶意安全性下，单一的从一个参与方接收秘密份额就不再安全，这是因为恶意参与方可以偏离协议，发送任意信息给重构的参与方。不过在复制秘密共享中，P_i 所需要的份额 x_{i+2} 同时被 P_{i-1}，P_{i+1} 拥有，因此就可以让 P_{i-1} 和 P_{i+1} 共同把 x_{i+2} 发送给 P_i，在三方诚实大多数设定下，P_{i-1} 和 P_{i+1} 至少有一个为诚实方，因此 P_i 只需要检查接受的两个数据是否为相同的，如果不同则输出 abort。相同即可按照半诚实下后续步骤进行。同理，在秘密公开的时候也需要分别向 P_{i-1}, P_{i+1} 发送信息，并进行一致性检验。恶意安全性下，每次重构需要两个参与方发送一个域上元素，而秘密公开需要每个参与方发送两个元素。

2.3.3 生成随机 RSS 秘密共享

生成随机秘密共享也是非常重要的函数，一般用 $\mathcal{F}_{rand}$ 来表示，其作用为生成一个随机元素的秘密共享。这里需要注意的是，生成过程以及生成后，参与方只能得知随机数的秘密共享而并不知道随机数的真实值。具体实现也很简单，只需要像生成零秘密共享一样，每个参与方 P_i 通过伪随机数生成器 $PRNG_{i-1,i}, PRNG_{i,i+1}$ 分别获得两个元素 $\rho_i, \rho_{i+1} \in \mathbb{F}_p$，这样 (ρ_0, ρ_1, ρ_2) 就构成了一个随机秘密共享，并且没有任何参与方知道具体的随机值。生成秘密共享没有通信开销。

与之相似的是直接共同生成随机数的函数，用 $\mathcal{F}_{coin}$ 来表示，其生成方式建立在 $\mathcal{F}_{rand}$ 上，获取随机秘密共享之后公开即可。注意这里不能够用伪随机数生成器替代，如果在协议初始化阶段设定一个公共的伪随机数生成器，调用 $\mathcal{F}_{coin}$ 时直接通过伪随机数生成器来获取随机数，这样导致的问题是：有些场景下，生成的随机数是需要确保不能够被任何一个参与方提前预知的，随机数只能由所有参与方共同协商出，如果使用伪随机数生成器则不能够满足这一场景。

第三章　恶意安全性下诚实大多数三方混合协议

本章节会详细介绍在隐语框架中所实现的混合协议的具体结构和技术。混合协议需要考虑到三方面问题：算术运算、布尔运算、秘密共享转化。只有三个方面均达到了恶意安全性，整个协议才能够满足恶意安全性。实际上，如果同或门和异或门在密态具有的同态性质，那么其实恶意安全性协议只需要关注秘密数与门运算的安全性即可。本章节介绍的协议考虑算术乘法、逻辑乘法、秘密共享在 $\mathbb{F}_p$ 和 $\mathbb{F}_2$ 之间转化的安全性。

3.1　算术运算

算术运算部分，本项目采用使用了 SPDZ 秘密共享方案[10] 的 Spdzwise 协议[9]，该协议在局域网条件下能够达到每秒计算超过 3,000,000 乘法运算的效率。前文已经提到，Spdzwise 协议的主要思想是用消息认证码来确保数据的完整性，每个数据 $[\![x]\!]$ 对应一个消息验证码 $[\![mac]\!]$，并且满足 $[\![mac]\!] = [\![x]\!] \cdot [\![\Delta]\!]$，其中 Δ 为全局 MAC 密钥并且以 share 形式存储在各个参与方。实际上，Spdzwise 协议建立在半诚实协议的基础上，很多操作都是使用半诚实协议运行，最终需要对所有半诚实的乘法运算进行一致性校验。

3.1.1　协议初始化

协议的初始化阶段，两两参与方之间初始化可认证的、安全的通信信道，并且设定一个广播信道。随后两两参与方之间约定伪随机数生成器密钥，并用来初始化伪随机数生成器 $PRNG_{i,j}$。然后，各参与方初始化半诚实协议，并调用半诚实协议的 $\mathcal{F}_{rand}$ 获取全局 MAC 密钥的秘密共享 $[\![\Delta]\!]$。协议还需要初始化一个缓冲区，用于后面存储需要校验的数据-消息认证码二元组。

3.1.2　秘密输入

对于任意一个输入 x_i，首先调用半诚实协议的 $\mathcal{F}_{input}$ 来获取 $[\![x_i]\!]$，随后调用半诚实协议的 $\mathcal{F}_{mult}$ 计算 $[\![x_i]\!] \cdot [\![\Delta]\!]$ 来得到 $[\![mac_{x_i}]\!]$，这就构成了一个 SPDZ 秘密共享，其构成为 $([\![x_i]\!], [\![mac_{x_i}]\!])$。此处乘法是半诚实的，因此需要将 x 的 SPDZ 秘密共享存入缓冲区，以后续检测。秘密输入的通信量包含一次半诚实协议秘密输入的通信量，和一次半诚实协议乘法的通信量。

3.1.3　加法与乘法

前文（2.2.2 节）提到，SPDZ 可认证秘密共享方案具有加法和常数乘法的同态性，这里只展开描述常数与秘密数的加法。假设拥有秘密数 x 的秘密共享 $[\![x]\!]$ 和常数 c，现在想要计算 $[\![x+c]\!]$。展开来看的话，需要计算 $\Delta(x+c)$ 和 $x+c$ 的复制秘密共享，对于 $x+c$ 的复制秘密共享前文已经介绍，对于 $\Delta(x+c)$ 的秘密共享，需要计算出 Δc 才能

够计算，这里显然 Δ 是以秘密共享的形式存在各参与方的，因此可以首先调用 RSS 的常数乘法方案，计算出 $c[\![\Delta]\!] = [\![\Delta c]\!]$，然后计算 $[\![\Delta(x+c)]\!] = [\![\Delta x]\!] + [\![\Delta c]\!]$ 即可。

对于两个秘密数的乘法，假设由两个秘密数 x 和 y，其 SPDZ 可认证秘密共享分别为 $([\![\Delta x]\!], [\![x]\!])$ 和 $([\![\Delta y]\!], [\![y]\!])$，目标是计算 $([\![\Delta xy]\!], [\![xy]\!])$。对于数据部分，可以直接调用半诚实协议的 $\mathcal{F}_{mult}$ 计算 $[\![xy]\!] = [\![x]\!][\![y]\!]$，而对于消息认证码部分，可以向 $\mathcal{F}_{mult}$ 传入 $[\![\Delta x]\!]$ 和 $[\![y]\!]$，这样就得到了 $[\![\Delta xy]\!] = [\![\Delta x]\!][\![y]\!]$。当然，这里所涉及到的两次半诚实乘法也要存入缓冲区。Spdzwise 的秘密数乘法通信开销为两次半诚实协议乘法通信开销。

3.1.4 使用“零校验”检验数据完整性

所有计算完成后，需要对缓冲区中保存的所有 SPDZ 可认证秘密共享进行检查。此时拥有 $([\![\Delta z_k]\!], [\![z_k]\!])_{k=1}^{N}$，参与方首先调用 $\mathcal{F}_{coin}$ 生成 N 个随机数，设为 $\alpha_1, \ldots, \alpha_m$。随后各个参与方本地计算：

$$[\![u]\!] = \sum_{k=1}^{N} \alpha_k \cdot [\![\Delta z_k]\!], [\![w]\!] = \sum_{k=1}^{N} \alpha_k \cdot [\![z_k]\!]$$

最后，调用 $\mathcal{F}_{mult}$，计算 $[\![T]\!] = [\![u]\!] - \mathcal{F}_{mult}([\![\Delta]\!], [\![w]\!])$，在没有恶意攻击者攻击的情况下，$T$ 的值应为 0。然而这里并不能直接公开 T 的值，原始论文[9] 中提供了一种零校验，即检查某个秘密共享 $[\![x]\!]$ 是否为零的方法：

- 各参与方调用 $\mathcal{F}_{rand}$ 来获取随机秘密共享 $[\![r]\!]$。
- 各参与方调用半诚实安全协议下 $\mathcal{F}_{mult}([\![x]\!], [\![r]\!]) = [\![xr]\!]$
- 各参与方公开 $[\![xr]\!]$，如果 $xr = 0$，那么认为 $x = 0$

这样，通过零校验的方法对 $[\![T]\!]$ 进行检查，即可判断数据是否被恶意攻击者篡改。

3.1.5 算术截位运算

Spdzwise 并没有提供截位运算协议，这里本项目选择 ABY^3[7] 协议中的截位方案（5.1.2 节，Share Truncation Π_{trunc2}），这种方案可以在一轮通信内完成截位运算。该截位运算协议需要在离线阶段生成“截位数对”（Truncation Pair），形式为 $([\![r]\!]_{\mathbb{F}_p}, [\![r']\!]_{\mathbb{F}_p})$，其中 $r = r'/2^d$。

生成 Truncation Pair 的思想是，假设 $[\![r]\!]_{\mathbb{F}_p} = (r_0, r_1, r_2)$，$[\![r']\!]_{\mathbb{F}_p} = (r'_0, r'_1, r'_2)$，各个参与方随机先随机生成 $[\![r']\!]_{\mathbb{F}_2}$，并直接截位得到 $[\![r]\!]_{\mathbb{F}_2}$，随后各参与方随机生成 $[\![r_1]\!]_{\mathbb{F}_2}$，$[\![r_2]\!]_{\mathbb{F}_2}$，$[\![r'_1]\!]_{\mathbb{F}_2}$，$[\![r'_2]\!]_{\mathbb{F}_2}$，注意，这里其实是 r 的秘密共享的秘密共享，由于参与方 P_i 拥有 r_i, r_{i+1} 和 r'_i, r'_{i+1}，因此可以把 r_1, r'_1 公开给 P_0, P_1，把 r_2, r'_2 公开给 P_1, P_2，最后各方调用二进制减法电路计算

$$[\![r_0]\!]_{\mathbb{F}_2} = [\![r]\!]_{\mathbb{F}_2} - [\![r_1]\!]_{\mathbb{F}_2} - [\![r_2]\!]_{\mathbb{F}_2}$$
$$[\![r'_0]\!]_{\mathbb{F}_2} = [\![r']\!]_{\mathbb{F}_2} - [\![r'_1]\!]_{\mathbb{F}_2} - [\![r'_2]\!]_{\mathbb{F}_2}$$

并把结果公开给 P_0, P_2，这样，各个参与方就拥有了足够的 $[\![r]\!]_{\mathbb{F}_p} = (r_0, r_1, r_2)$，$[\![r']\!]_{\mathbb{F}_p} = (r'_0, r'_1, r'_2)$。

线上阶段，拥有了 Truncation Pair 之后，假设想要计算 $[\![x]\!]_{\mathbb{F}_p} = [\![x']\!]_{\mathbb{F}_p}/2^d$ 参与方共同计算 $[\![x' - r']\!]$ 并公开，随后 $[\![x]\!]_{\mathbb{F}_p} = [\![r]\!]_{\mathbb{F}_p} + (x' - r')/2^d$ 就得到了截位后的结果。

3.1.6 安全性分析

下面分析 Spdzwise 协议是如何抵抗恶意攻击者攻击的。Spdzwise 协议建立在半诚实协议的基础上，乘法计算都先要执行一种不能完全抵抗恶意攻击者攻击的乘法协议。但是协议要求对于这种乘法运算，恶意攻击者最多只能做到“加法攻击”（up to Additive Attack），意思是说如果目标是计算 $x \cdot y$，攻击者攻击后只能做到能够改变结果为 $x \cdot y + d$，其中 d 是由攻击者选择，并且是独立于 x 和 y 的。

在“Multiplication up to Additive Attack”的情况下，以复制秘密共享乘法为例。假设想要计算秘密数 x 和 y 的乘积，如果攻击者想要改变其结果为 $xy + d$，对于数据部分，攻击者计算出 z_i 与零秘密共享之后，只需发送 $z_i + \sigma_i + d$，秘密共享对应的秘密数就变成了 $z' = z_0 + z_1 + z_2 + d = z + d$。然而，在 SPDZ 可认证秘密共享下，攻击者还需要伪造 $z + d$ 对应的消息认证码，即 $\Delta(z + d)$，那么攻击者就需要在消息认证码部分计算出 z_i 后，增加一个 $\Delta \cdot d$ 的偏移，然而攻击者无法得知 Δ 的任何有关信息，此时伪造出正确信息的概率只有 $1/|\mathbb{F}_p|$，当 $|\mathbb{F}_p|$ 足够大时（本毕设采用域大小为 61-bit 梅森素数域），攻击者攻击成功的概率很低。

在最终的验证过程中，首先，校验环节两次调用半诚实协议下的 $\mathcal{F}_{mult}$，但是这两次并不能递归的去被检查，这里存在着被恶意攻击者攻击的可能。然而，如果考虑该协议达到的安全性为带终止的安全（Security with Abort），这种攻击是能够被容忍的，因为攻击者只有可能将正确的数据改写为错误的，从而导致本应正常执行的协议未完成提前终止，并没有影响到协议的隐私性与正确性，因此这在带终止的安全下是允许的。

其次，零校验过程中通过判断 xr 是否为 0 来确定 x 是否为零，这种做法要求协议所使用的数学结构为域，在域中不存在零因子，这也就说明唯一能使得 $r \neq 0$ 但 $xr = 0$ 的情况就是 $x = 0$，出错的概率只有 $1/|\mathbb{F}_p|$，如果选择一个大的域依旧是安全的。

最后，根据原始论文[9] 引理 4.2，如果攻击者对任意一个乘法进行了攻击，那么检验完整性过程中 $T = 0$ 的概率小于 $2/|\mathbb{F}_p|$，这种误差是在检验完整性中，随机选取系数 α_k 不当导致攻击者攻击产生的误差被消除导致的。

综上所述，在消息认证码与原始数据不匹配的情况下，检验依旧通过的概率小于 $3/|\mathbb{F}_p|$。

3.1.7 关于 Spdzwise 协议的扩展

Spdzwise 协议建立在半诚实安全的协议之上，实际上，通过上文所描述的细节即可看出，任何满足以下三个要求的秘密共享方案都能够应用到 Spdzwise 协议当中：

- 同态性：能够满足常数与秘密数加法、秘密数与秘密数加法、以及常数与秘密数乘法具有同态性。
- 可打开性：对于秘密分享 $[\![x]\!]$，参与方可以重构或公开秘密值 x。
- 隐私性：攻击者无法从秘密分享 $[\![x]\!]$ 中获取到有关秘密数 x 的任何信息。

当然除此之外，还应该考虑使用域结构以及 $3/|\mathbb{F}_p|$ 误差问题，满足这些要求的秘密分享，配合一种恶意攻击者只能进行加法攻击的乘法协议，即可嵌入 Spdzwise 协议当中。

3.2 布尔运算

最早姚期智提出的混淆电路方案[18] 是最早对于安全多方计算的解决方案，其针对逻辑电路的每个导线进行编号，并对每个门设置编码表进行估值，后续出现了大量针对混淆电路方案的改进以及基于秘密共享的方案。对比来看，基于混淆电路的方案通信开销更低，因为整个协议只需要一次通信；而基于秘密共享方案通常具有较好的扩展性，能够适应较大规模的参与者网络。因此在布尔运算部分，本项目采用 FLNW17[11] 中提出的协议，这是一种使用秘密共享、基于 Beaver[13] 的协议，并且采用了切片选择方法[14] 来生成 Beaver 三元组。该协议使用了一种离线-在线（Offline-Online）思想，协议分成了需要生成辅助数据的离线阶段，和实际进行运算的在线阶段，离线阶段不依赖于参与方的输入，而在线阶段依赖于参与方输入。这种思想在后文秘密转化中也会提到。

由于 FLNW17 使用了 $\mathbb{F}_2$ 上的复制秘密共享方案，因此秘密输入输出、加法、常数乘法等与 2.3 节提到的子协议完全相同，此处不再赘述。在本节主要分析基于 Beaver 三元组的二进制与门运算，以及 Cut-and-Choose 方案。

3.2.1 基于 Beaver 三元组的二进制与门运算

Beaver 三元组，是由一组满足关系 $[\![a]\!] \cdot [\![b]\!] = [\![c]\!]$ 的三元组 $([\![a]\!], [\![b]\!], [\![c]\!])$ 构成，其中 a 和 b 是随机选取，c 满足了 $c = a \cdot b$ 的关系。在二进制与门中，每次与运算均消耗一个 Beaver 三元组。

在乘法阶段，假设已经获取了一组合法的 Beaver 三元组 $([\![a]\!], [\![b]\!], [\![c]\!])$，现在考虑乘法的两个输入为 x, y，其对应的秘密共享是 $[\![x]\!], [\![y]\!]$。计算 xy 的秘密共享，可以分为如下几步：

- 各个参与方本地计算 $[\![x - a]\!] = [\![x]\!] - [\![a]\!]$，并公开 $d = x - a$ 的值。

- 各个参与方本地计算 $[\![y-b]\!]=[\![y]\!]-[\![b]\!]$，并公开 $e=y-b$ 的值。
- 各个参与方本地计算

$$[\![xy]\!]=de+d[\![y]\!]+e[\![x]\!]+[\![c]\!]$$

这便得到了 xy 的秘密共享。

第三步中的等式之所以成立，是因为

$$\begin{aligned}xy&=(x-a+a)(y-b+b)\\&=(d+a)(e+b)\\&=de+db+ae+ab\\&=de+db+ae+c\end{aligned}$$

而这时 d 和 e 均已经被打开，因此式中所有运算均可以离线完成。上述所有运算在 $\mathbb{F}_2$ 同样成立。若不考虑生成 Beaver 三元组，乘法阶段通信开销为两个域上元素的大小。

3.2.2 “切片选择”方法生成 Beaver 三元组

可以看出，Beaver 方案的在线阶段并没有太大的开销，而把问题抛给了离线阶段如何生成合法的 Beaver 三元组。Chaum 等人[14] 用“切片选择”思想实现了一种盲签名方案，而目前 FLNW17[11] 所采用的基于“切片选择”生成 Beaver 三元组的方法是目前最高效的方案。

“切片选择”（Cut-and-Choose）方法的核心在于“Sacrifice”，即牺牲一部分的数据来换回另一部分数据的正确性。首先，各个参与方随机选择 N 组 $[\![a_i]\!],[\![b_i]\!]$，并使用半诚实安全性协议的乘法来计算 $[\![c_i]\!]=\mathcal{F}_{mult}([\![a_i]\!],[\![b_i]\!])$。接下来，Cut-and-Choose 方案需要将这些随机的、待验证的三元组进行随机排列，打乱原本的顺序，并共同公开前 C 个三元组，如果前 C 个三元组当中检测到某组 $a_i\cdot b_i\neq c_i$，那么协议就会终止。如果验证通过，则继续将剩余未打开的 $N-C$ 个三元组分成一个个的“桶”（Bucket），每个桶内部会牺牲其他的三元组来确保第一个三元组的正确性。最终，每个桶输出第一个三元组，“切片选择”结束。

这个过程中，最核心的部分在于在每个桶中，如何“牺牲其他的三元组来确保第一个三元组的正确性”。FLNW17 文中协议 2.24（PROTOCOL 2.24 Triple Verif. Using Another Without Opening）提到了一种方法，假设用三元组 $([\![a]\!],[\![b]\!],[\![c]\!])$ 去验证三元组 $([\![x]\!],[\![y]\!],[\![z]\!])$，首先，每个参与方本地计算 $[\![\rho]\!]=[\![x]\!]+[\![a]\!],[\![\sigma]\!]=[\![y]\!]+[\![b]\!]$ 并公开，最后，各参与方计算

$$[\![T]\!]=[\![z]\!]+[\![c]\!]+\sigma\cdot[\![a]\!]+\rho\cdot[\![b]\!]+\rho\cdot\sigma$$

并检查 T 是否等于 0 即可完成，因为在 $\mathbb{F}_2$ 上，有

$$
\begin{aligned}
& z + c + \sigma \cdot a + \rho \cdot b + \rho \cdot \sigma \\
=& z + c + (y + b) \cdot a + (x + a) \cdot b + (x + a) \cdot (y + b) \\
=& z + c + ay + ab + bx + ab + xy + xb + ya + ab \\
=& z + c + xy + ab \\
=& 0
\end{aligned}
$$

如果 $([\![a]\!], [\![b]\!], [\![c]\!])$ 是一组合法的 Beaver 三元组，$([\![x]\!], [\![y]\!], [\![z]\!])$ 是一组不合法的三元组，那么上述检查必然会输出错误。换句话说，若参与到检查的两个三元组正确性不一致，协议则会报错。

实际上，在三个参与方的复制秘密共享情况下，还有一种优化通信量的检查方法。例如，现在拿到了 $[\![T]\!] = (T_0, T_1, T_2)$，其中参与方 i 拥有 (T_i, T_{i+1})，这里 FLNW17[11] 中提供了一种基于哈希的优化思路：若 $T_0 \oplus T_1 \oplus T_2 = 0$，那么在二进制运算上，参与方 P_i 自身拥有的两个秘密共享之和，应该与参与方 P_{i+1} 所拥有的 T_{i+2} 一致。因此，每个参与方 P_i 可以维护两个哈希对象，分别存储 $T_i \oplus T_{i+1}$ 和 T_{i+1}，最终得到两个哈希 h_i, h_{i+1} 并发送 h_i 给 P_{i+1}，这样，自身的 h_{i+1} 与从 P_{i-1} 收到的 h' 应该是相等的。这样，把原本一个批次的通信量降低为了一个哈希值的通信大小。

最优情况下，设 bucket size 为 B，那么对于每个 Beaver 三元组，需要生成 B 个三元组，并在牺牲阶段进行 $B-1$ 次验证，这样，每次验证需要 2 比特通信量，每个生成需要 1 比特通信量，使用切片-选择生成一个可信三元组的通信量就是 $3B-2$。在 $B=4$ 下，生成每个可信 Beaver 三元组的通信量为 10 比特。

3.2.3 安全性分析

在线阶段，由于秘密数 x 和 y 分别被 Beaver 三元组的随机数 a 和 b 掩盖，因此公开 $x-a$ 和 $y-b$ 的值不会泄露任何关于 x, y 的信息，后续计算也仅是本地运算。其次在“切片选择”阶段，前期的随机打乱操作确保了恶意攻击者不会提前根据位置信息精心构造三元组来绕过三元组检查，随后打开前 C 个三元组目的是确保参与“切片选择”的 N 个待检测三元组中不会存在过多的错误三元组。最后，如果想要绕过每个桶中的检查，则必须确保桶中的每个三元组均为错误的，并且是经过了精心构造的，然而这在随机打乱的前提下基本是不可能的。

想要深入分析恶意攻击者成功伪造一个错误 Beaver 三元组的概率，需要引入一个引理。

引理 3.1 设有 N 个桶，共有 $N \cdot B$ 个球，这些球中有 B 个为红色，其余都是黑色，红球中有一颗球编号为 1，其余均匀一致，所有黑球均匀一致。现在想要把这 $N \cdot B$ 个球随机、平均放进这 N 个桶，使得每个桶恰好有 B 个球，则这 B 个红球按照编号 1 在同

一个桶中的概率为

$$P = \frac{N}{\binom{N\times B}{B} \times B}$$

证明：因为桶内的位置是存在顺序的，因此可以把所有桶并排放置，并将内部位置看作是一维空间，其长度为 $N\cdot B$，那么这 $N\cdot B$ 在一维下的所有排列数量为 $\binom{N\times B}{B}\times\binom{B}{1}$，这其中正确的放置方法只有 N 种，对应红球所处的桶编号分别为 $1\ldots N$，因此成功放置的概率为

$$\frac{N}{\binom{N\times B}{B} \times \binom{B}{1}} = \frac{N}{\binom{N\times B}{B} \times B}$$

借助引理 3.1，可以立刻得出恶意攻击者想要成功伪造一个错误的 Beaver 三元组的概率。假设忽略“切片选择”方法中打开前 C 个 Beaver 三元组的过程，那么场景就是拥有 $N\cdot B$ 个待验证的 Beaver 三元组，攻击者需要将自己生成的 B 个错误的 Beaver 三元组放入同一个桶，并且因为这 B 个错误三元组是精心设计以用于绕过安全检查，需要确保要输出的 Beaver 三元组位于桶的第一个位置，这对应了引理 3.1当中红球编号的场景。当设置 $N = 2^{20}, B = 3$，此时攻击者攻击成功的概率小于 $1/2^{40}$。

3.2.4 关于 Beaver 协议的扩展

就像 Spdzwise 协议一样，也可以抽象出满足 Beaver 协议的秘密共享方案所需要具备的性质：

- 可打开性，即各方公布自己的秘密份额后，各个参与方可以恢复秘密值。
- 同态性，包括加法同态和常数乘法同态。
- 隐私性，即秘密份额 $[\![x]\!]$ 不会泄露任何与秘密值 x 相关的信息。

除此之外，还应确保这种秘密共享方案能够生成符合条件的 Beaver 三元组。满足上述条件的秘密共享方案均能够嵌入到 FLNW17 方案中。

3.3 秘密共享转化

在安全多方计算领域，秘密共享在算术与二进制之间的转换是一项重要的研究内容。早期的秘密共享转化（如 ABY^3[7]）采用了不经意传输和 PPA 电路的方法，随后出现了类似 Beaver 方案离线-在线模式的方法[16][15]，近年来，出现了一批基于多项式插值和快速傅里叶变换技术的方案[19]，在通信和计算开销方面实现了优化。本毕设项目采用了 Daniel 等人基于 EdaBit 的转化方法，这种方案在 Online-Offline 模式里拥有很好的效率，根据原文描述，恶意安全性诚实大多数场景下，每秒可以生成 156 个 128-bit 域上的 EdaBits。

3.3.1 基于“切片选择”的 EdaBit 生成方法

EdaBit 的概念源自于 daBit（double-shared authenticated Bit），假设协议运行的底层数学结构为 $\mathbb{F}_p$，那么一个 daBit 其实就是一个随机比特 $r \in \mathbb{F}_2$ 在算术和二进制域上秘密共享的二元组，即 $(\llbracket r \rrbracket_{\mathbb{F}_p}, \llbracket r \rrbracket_{\mathbb{F}_2})$。Daniel 等人做出了改进，将随机数 r 的范围扩展到了更大的域 $\mathbb{F}_p$ 上（也可以是环 $\mathbb{Z}_{2^k}$），设这个环或域的大小为 m-bits，那么一个 Extended daBit 的结构就变成了

$$\{\llbracket r \rrbracket_{\mathbb{F}_p}, \llbracket r_0 \rrbracket_{\mathbb{F}_2}, \llbracket r_1 \rrbracket_{\mathbb{F}_2}, \ldots, \llbracket r_{m-1} \rrbracket_{\mathbb{F}_2}\}$$

论文中同样提出了如何采用“切片选择”思想来满足恶意安全性生成可信 EdaBits，本项目只考虑域上的操作。

全加器电路与比特转化 为了能够生成随机 EdaBit，需要做两个前置工作：安全的全加器电路与比特转化。安全的全加器电路，即给定两个布尔秘密共享形式的比特流 $\{\llbracket x_0 \rrbracket_{\mathbb{F}_2}, \ldots, \llbracket x_{m-1} \rrbracket_{\mathbb{F}_2}\}$ 和 $\{\llbracket y_0 \rrbracket_{\mathbb{F}_2}, \ldots, \llbracket y_{m-1} \rrbracket_{\mathbb{F}_2}\}$，计算秘密共享比特流形式的结果，并且该方法需要能够达到恶意安全性。实际上，全加器电路的实现有很多，但是对于安全多方计算来说，计算开销集中在计算与门，因此应尽量选择与门数量少的全加器电路。论文中提出一种电路，对于每个比特位，有：

$$c_{i+1} = c_i \oplus ((x_i \oplus c_i) \wedge (y_i \oplus c_i)), \forall i \in 0, \ldots, m-1$$

其中 $c_0 = 0$。这样的话，输出就可以定义为 $z_i = x_i \oplus y_i \oplus c_i, i = 1, \ldots, m-1$，最后一位输出定义为 $z_m = c_m$。这样的全加器电路，每个比特位需要一个与门运算，想要达到恶意安全性，只需要确保与门运算的恶意安全性即可。

比特转化（Bit Injection），目标是将一个布尔秘密分享的比特转化为算术秘密分享下的比特。ABY^3[7] 中提供了使用三方不经意传输的做法，当然也可以使用 daBit 来进行比特转化。

生成随机 EdaBit 假设域 $\mathbb{F}_p$ 的大小为 m-bits，生成随机 EdaBit 的主题思想是各参与方先本地生成 Edabit，然后进行合并。首先，参与方 P_i 随机生成比特 $r_0, \ldots, r_{m-1} \in \mathbb{F}_2$，随后，参与方 P_i 计算 $r = \sum_{i=0}^{m-1} r_i 2^i$ 并将 r 作为输入，通过秘密输入生成秘密共享并分发给各个参与方。现在，各个参与方拥有 $\{\llbracket r_i \rrbracket_{\mathbb{F}_p}, \llbracket r_{i,0} \rrbracket_{\mathbb{F}_2}, \ldots, \llbracket r_{i,m-1} \rrbracket_{\mathbb{F}_2}\}$，其中 $i = 1, \ldots, n$，n 为参与方的数量。随后，各参与方需要“纵向的”将这些私有 EdaBits 进行叠加。算术部分只需要计算 $\llbracket r' \rrbracket_{\mathbb{F}_p} = \sum_{i=1}^{n} \llbracket r_i \rrbracket_{\mathbb{F}_p}$ 即可，然而对于二进制部分，二进制加法需要通过全加器电路完成，n 个 m 比特数据通过二进制进位全加器计算后，得到的结果长度为 $m + \log n$ 比特，记为 $(\llbracket b_0 \rrbracket_{\mathbb{F}_2}, \ldots, \llbracket b_{m+\log(n)-1} \rrbracket_{\mathbb{F}_2})$，不过由于算术运算所叠加的随机数是在 $\mathbb{F}_p$ 上进行，而对于二进制电路来说，如果直接截位取最低 m 比特位，其实相当于是在 Z_{2^m} 上进行的运算。为了解决这一问题，需要每个参与方调用比特转化，将第 m 比特之后更高比特位全部打开，即将 $\llbracket b_j \rrbracket_{\mathbb{F}_2} \to \llbracket b_j \rrbracket_{\mathbb{F}_p}, j = m, \ldots, m + \log(n) - 1$，然后把超

出部分截位的差异在算术部分消去，也就是最后计算

$$[\![r]\!]_{\mathbb{F}_p} = [\![r']\!]_{\mathbb{F}_p} - 2^m \sum_{j=0}^{\log(n)-1} [\![b_{j+m}]\!]_{\mathbb{F}_p} 2^j.$$

这样，就得到了共同计算出的 EdaBit$\{[\![r]\!]_{\mathbb{F}_p}, [\![b_0]\!]_{\mathbb{F}_2}, [\![b_1]\!]_{\mathbb{F}_2}, \ldots, [\![b_{m-1}]\!]_{\mathbb{F}_2}\}$

使用“切片选择”技术筛选 为了达到恶意安全性，参与方之间还需要进行一次切片选择来确保输出的正确性。其思想仍然是打乱所有输入，打开前 C 个进行校验，分组，使用 “Sacrifice” 的思想进行检查。这里重点说明如何进行桶内检查。假设有用于检查和牺牲的 EdaBit$\{[\![r]\!]_{\mathbb{F}_p}, [\![r_0]\!]_{\mathbb{F}_2}, [\![r_1]\!]_{\mathbb{F}_2}, \ldots, [\![r_{m-1}]\!]_{\mathbb{F}_2}\}$，$\{[\![s]\!]_{\mathbb{F}_p}, [\![s_0]\!]_{\mathbb{F}_2}, [\![s_1]\!]_{\mathbb{F}_2}, \ldots, [\![s_{m-1}]\!]_{\mathbb{F}_2}\}$：

- 首先，计算两者算术上的和，即 $[\![r+s]\!]_{\mathbb{F}_p} = [\![r]\!]_{\mathbb{F}_p} + [\![s]\!]_{\mathbb{F}_p}$
- 计算两者在二进制上的和，即调用全加器电路将 $\{[\![r_0]\!]_{\mathbb{F}_2}, \ldots, [\![r_{m-1}]\!]\}$ 与 $\{[\![s_0]\!]_{\mathbb{F}_2}, \ldots, [\![s_{m-1}]\!]\}$ 求和，得到结果 $\{[\![c_0]\!]_{\mathbb{F}_2}, \ldots, [\![c_m]\!]\}$
- 将最高比特位的差异消去，把最高比特位的布尔秘密共享转化为算术秘密共享，即 $[\![c_m]\!]_{\mathbb{F}_2} \to [\![c_m]\!]_{\mathbb{F}_p}$
- 计算消除差异的算术部分，$[\![c']\!]_{\mathbb{F}_p} = [\![r+m]\!]_{\mathbb{F}_p} - 2^m \cdot [\![c_m]\!]_{\mathbb{F}_p}$，公开 c' 和对应比特流 $c_0, \ldots, c_{m-1}$，检查比特是否对应于 c'

  所有检查通过，则生成了一个批次的 EdaBits。

3.3.2 使用 EdaBit 进行秘密共享转化

假设现在有秘密数据 x 对应的算术秘密共享 $[\![x]\!]_{\mathbb{F}_p}$，现在需要将其转化为二进制秘密共享向量（B2A），即 $[\![\vec{x_i}]\!]_{\mathbb{F}_2}$，目前学界常见的方案是这样：考虑到利用上 EdaBit $\{[\![r]\!]_{\mathbb{F}_p}, [\![b_0]\!]_{\mathbb{F}_2}, [\![b_1]\!]_{\mathbb{F}_2}, \ldots, [\![b_{m-1}]\!]_{\mathbb{F}_2}\}$，各参与方计算 $[\![x-r]\!]_{\mathbb{F}_p}$ 并公开，随后在明文上计算 $x-r$ 的比特分解并转化为布尔秘密共享 $[\![b_i]\!]_{\mathbb{F}_2}$，最后使用全加器电路，在二进制上计算 $x-r$ 与 r 的和。

这种方案在环上可行，但是在域上会存在一个问题：假设计算 $x-r$ 时发生了负溢出，即计算出的结果为 $x-r+p$，在这种情况下使用全加器电路计算与 r 的和，得到的会是 $x+p-2^k$，与目标不符。不过，考虑到机器学习过程中数据的范围，假设数据 x 的大小在 30 比特，生成 EdaBit 中随机数大小在 60 比特，那么此时计算 $x+r$ 则不会发生正溢出，后续运算中只需要用全加器电路将 $x+r$ 与 r 的二进制补码相加即可。

在布尔秘密共享转化到算术秘密共享（A2B）过程中，事情会变得简单很多。首先，在布尔电路上计算 $x+r$ 的值，并进行公开。需要注意的是，因为这里是二进制上的算术运算，其打开之后不会受到模运算影响，会得到 $x+r$ 的真实值。此时再将 $x+r$ 进行秘密共享并减去算术秘密共享 $[\![r]\!]_{\mathbb{F}_p}$，即可得到算术秘密共享 $[\![x]\!]_{\mathbb{F}_p}$。

3.3.3 安全性分析

采用同样的分析手段，可以对生成 EdaBits 所使用的“切片选择”方法进行概率计算。此处直接引用原始论文的结论：

$$Pr[\mathcal{A}\ passes\ BucketCheck] \leq \frac{(B-1)!}{(NB-(B-2))^{B-1}} \leq 2^{-s},\ for\ B \geq 3.$$

如果采用 $B = 4$，每批次生成 EdaBit 数量大于 10332，此时能够达到 40-bit 安全强度。

在秘密共享转化方案中，存在一个需要注意的安全问题：公开了 $x + r$ 的值。假设已知 x 和 r 的最大取值为 t_{max}，而公开的 $x + r$ 的值为 $2 * t_{max}$，那么可以立刻推断出 $x = r = t_{max}$，这样就无法满足了信息论安全性。为了尽量避免这种情况的发生，需要让 r 远大于 x，如前文所述。这种秘密共享转化方案妥协了计算过程中的数据大小，但是如果按照原始方案，需要额外进行一次 x 与 r 的大小比较，以此来判断是否存在溢出，并在打开 $x + r$ 后消去溢出产生的影响，或者实现一种模运算的加法电路。这两种方案均包含开销很大的操作，考虑到机器学习算法本身的数据特征，本项目选择了在数据范围上的妥协。

第四章 “隐语”中实现恶意安全性混合协议

本节会主要介绍章节 3中提到的混合协议具体是如何实现在“隐语”框架的。首先，本文会介绍一些上层关于隐语中密态设备 SPU（Secure Processing Unit）的理念，然后会探讨在实现过程中采用的优化策略以及性能提升方法，最后详细阐述在“隐语”框架下实现恶意安全性混合协议的具体步骤和方法。

4.1 密态设备“SPU”

SPU（Secure Processing Unit）是 SecretFlow 项目的核心组件之一，旨在提供具有可证明安全性的安全计算服务，并在设计时考虑到了轻松集成到其他分布式系统中的需求。SPU 本身作为一种设备，外部用户只需要像使用 CPU 一样传入数据和代码即可进行计算，不同的是在 SPU 中（见图 4-2 ）数据和代码可能是来自不同用户的，这与常见的 CPU 模式（见图 4-1 ）不同。

在 SPU 内部，实际上是由多个参与方协同计算，同时，内部参与计算的参与方可以与外部数据提供方有交集。不过目前应用中常见的仍然是 Client-Server 模式，即用户提供数据，而服务端提供隐私计算（见图 4-3 ）。

在 API 层面，SPU 上层对接常见的支持 XLA 的前端（例如 TensorfloW, JAX, PyTorch), 通过 SPU 编译器将其编译为 SPU IR, 最终调用底层接口来执行计算（见图 4-4 ）。目前 SPU 底层支持的 MPC 协议包括 ABY^3[7]、SPD$\mathbb{Z}_{2^k}$ 的简化版本[8] 以及 Cheetah[20]。拿二元运算举例，二元运算可能包含明文与明文、明文与密文、密文与密文之间的运算，但是在 SPU 的 HLO 层并不关注，HLO 负责处理整数运算与浮点数运算的类型，告诉 HAL 层两个数据是进行整数运算还是浮点数运算，而在 HAL 层会具体分析两种数据具体是明文还是密文，并分类到 ss (secret and secret)、sp (secret and plain)、pp (plain and plain)。最后在 MPC 层，会具体判断秘密数或者明文是在 Z_n 上还是 Z_2 上，并定义到具体的内核。

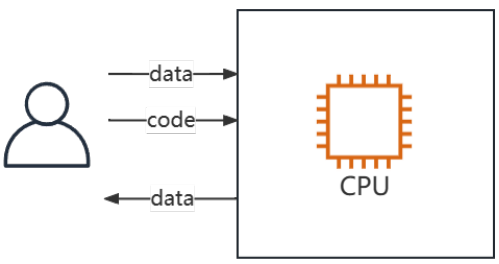

图 4-1 用户与 CPU 交互

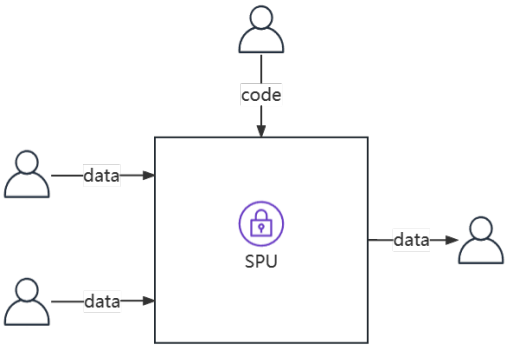

图 4-2 多用户与 SPU 交互

图片来源：https://www.secretflow.org.cn/docs/spu/en/development/basic_concepts.html

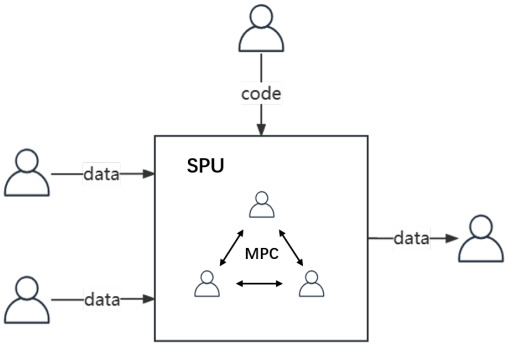

图 4-3 SPU 内部

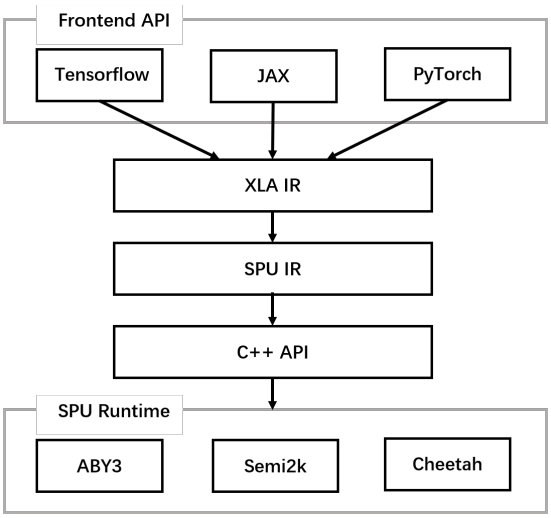

图 4-4 API 框架

图片来源：https://www.secretflow.org.cn/docs/spu/en/development/basic_concepts.html

4.2 在“隐语”中实现恶意安全性混合协议

本节会介绍在“隐语”中实现恶意安全性混合协议的具体方法。在这之前，需要先介绍两种核心优化策略，这两种策略贯穿了整个实现的全部。后文会介绍“SPU”框架中的一些代码思想与协议实现。本工作所有代码量约为 5775LoC (lines of code)，所涉及代码详见https://github.com/DuanYuFi/spu/tree/ddae0b8f49ac071a3af4023fa05ff88a64939f5f。

4.2.1 核心优化策略

除了隐语框架外，许多框架（如 MP-SPDZ）都使用了两个关键的优化策略：线程级并发（Thread-Level Concurrency）和单指令多数据并行（SIMP, Singie-Instruction Multiple-Data Parallelism）。

线程级并发 线程级并发优化在当今的大型项目中非常常见。一个工作任务可以被分解为多个步骤，这些步骤可以用有向无环图（DAG）来表示。在这个图中，同一层中的节点之间没有依赖关系，因此可以使用多个线程并发处理这些节点任务。在 SPU 中，主要针对与循环轮数无关的操作使用线程级并发，将所有循环都采用多线程并发执行，以最大程度地利用 CPU 性能。线程级并发的优势在于充分利用了现代多核处理器的并行能力。通过将任务分配给不同的线程，可以同时执行多个任务，从而提高整体的计算速度。然而，线程级并发也需要考虑线程之间的同步和数据访问冲突，以避免出现竞争条件和死锁等并发问题。

单指令多数据并行 单指令多数据并行（SIMD）是一种优化技术，允许一条指令同时处理多个数据元素。当需要对大量数据进行相同的操作时，SIMD 可以显著提高计算的效率。它适用于那些可以被向量化的任务，其中相同的操作可以同时应用于数据集合中

的多个元素。例如，在计算向量 $\vec{x} = (x_0, x_1, \ldots, x_{n-1})$ 和 $\vec{y} = (y_0, y_1, \ldots, y_{n-1})$ 的内积时，每对数据的乘法运算都需要进行通信和计算。传统上，这需要 n 次乘法操作，产生 $2n$ 个通信量和 n 次通信。然而，由于这些数据之间不存在依赖关系，可以在一轮通信中将所有需要交互的数据发送完毕。这种优化方法不会改变理论通信量的开销，但可以降低通信轮数、函数调用等环节的开销，从而提高整体性能。单指令多数据并行可以通过 SIMD 指令集（如 SSE、AVX 等）来实现，在支持这些指令集的处理器上，可以同时处理多个数据元素，减少了指令级别的开销。然而，要充分发挥 SIMD 的优势，需要设计算法和数据结构以适应向量化操作，并合理利用硬件的特性。

4.2.2 代码结构与思想

SPU 当中，所有的协议均抽象为一个名为 $Object$ 的类，类中包含了“状态”（$State$）和“内核”（$Kernel$），其中 $State$ 涵盖了这个协议中涉及到状态的成员变量，例如通信模块、伪随机数生成器、协议运行所依赖的数学结构等。在本次毕设中，前文所提到的 SPDZ 秘密共享密钥、保存需要检查的数据等也属于一种 $State$。而 $Kernel$ 则是用于存储一个协议应该支持的计算单元，例如明文与算术秘密共享的加法、算术秘密共享之间的乘法、截位操作等。

SPU 当中存在很多运行时动态类型。例如某两个数据分别占用 32 比特空间，但是他们相减之后，也许结果只有 8 比特大小。此时如果使用 $uint8\_t$ 类型进行存储，显然要比使用 32 比特空间的 $uint32\_t$ 更节约内存，因此为了统一化管理这种动态类型，SPU 中拥有一个完备的类型系统（Type System），并且其内置了一种支持动态类型的数组结构 $ArrayRef$，用于保存某个动态类型数据。实际上，这种动态结构并不需要存储太多的信息，只需要定义其结构，并给出该类型的大小，$ArrayRef$ 就可以合理分配空间为其存储，至于该类型的具体结构、理念等，$ArrayRef$ 并不关心。

另外一个值得注意的地方在于 SPU 的 Client-Server 模型（简称 C/S 模型）。无论有多少个参与方，服务端的数量是与协议绑定的。例如本项目实现了一个恶意安全性诚实大多数三方协议，那么就限制了服务端只能是三方负责计算，并且最多只能有一个腐败参与方。不过客户端的数量是任意的，客户端拥有数据，服务端负责计算。客户端发送给服务端的数据应该是已经分片的（就像复制秘密共享格式那样），数据交到服务端之后，服务端对秘密数据一无所知。当然一些恶意安全性的协议（例如本项目的 Spdzwise Field 协议）还要对输入数据进行认证，即将输入的秘密共享与全局消息认证码密钥的秘密共享相乘。当一切准备就绪后，数据便会以 $ArrayRef$ 的形式流动在各个 $Kernel$ 中。所有运算结束后，计算集群会把分片发送给客户端，客户端来重构数据。

为了做到这种 Client-Server 模型，SPU 当中还存在一个平行于 $Object$ 的类：IO。IO 负责了上述提到的数据分片与重组的操作，仅会在协议执行输入与输出时调用。其余过程中关于明文与密文数据的转化均通过调用 $P2A, A2P, P2B, B2P$ 来完成。

4.2.3 实现细节

整体架构 根据 4.2.2一节的介绍，想要创建一个完整协议，需要考虑到协议中的状态变量、运算单元两个部分。根据 SPU 中对 API 的设计要求，可以得到如下 Kernel 的需求清单：

名称	作用	名称	作用
RandA	生成随机秘密共享	P2A	将明文转化为算术秘密共享
A2P	将算术秘密共享转化为明文	P2B	将明文转化为布尔秘密共享
B2P	将布尔秘密共享转化为明文	NotA	对于输入 $[\![x]\!]$，计算 $[\![!x]\!]$
AddAP	明文与算术秘密共享的加法	AddAA	两个算术秘密共享的加法
MatMulAP	明文与算术秘密共享的矩阵乘法	MulAP	明文与算术秘密共享的乘法
MatMulAA	两个算术秘密共享的矩阵乘法	MulAA	两个算术秘密共享的乘法
LShiftA	秘密数算术左移	TruncA	秘密数算术右移
AndBP	布尔秘密共享与明文与运算	AndBB	两个布尔秘密共享与运算
XorBP	布尔秘密共享与明文异或运算	XorBB	两个布尔秘密共享异或运算
LShiftB	布尔秘密共享左移	RShiftB	布尔秘密共享右移
A2B	算术到布尔秘密共享转化	B2A	布尔到算术秘密共享转化

除此之外，根据本工作实现上的需求，还额外补充了下面的 Kernel：

名称	作用
P2ASH	半诚实协议下的明文转化为算术秘密共享
A2PSH	半诚实协议下的算术秘密共享转化为明文
MulAASH	半诚实协议下两个算术秘密共享的乘法
BitInject	将一比特布尔秘密共享转化为算术秘密共享

状态量部分，除去 SPU 已经包含的 $Communicator$（通信相关）、$PrgState$（伪随机数相关）等状态外，本工作新增了三个状态：

1. $SpdzWiseFieldState$，即算术运算部分相关状态，包含了计算所在域的信息（本文使用 61-bit 梅森素数域）、全局消息认证码密钥、已存储待检验的 SPDZ 秘密共享二元组、验证函数、生成算术运算所需的 truncation pair 函数。在计算设计中，每次乘法算术运算、秘密共享输入都会将结果的 SPDZ 秘密共享存储在 $SpdzWiseFieldState$ 的缓冲区当中，只有当缓冲区大小到达一定门限大小后才会执行检查。

2. $BeaverState$，即布尔运算部分相关状态，包含了生成布尔电路计算所需的关键信息。在这个状态中，主要包含了使用” 切片-选择” 方式生成 Beaver 三元组的方法。Beaver 三元组用于实现恶意安全性的布尔电路与门计算，用于计算和验证布尔电

路的输出。$BeaverState$ 中还包含其他与切片选择的相关参数，如 $batchsize, bucketsize$ 等。

3. $EdabitState$，即秘密共享转化相关状态，包含了生成 EdaBit 的关键信息。EdaBit 用于将秘密数据在算术与布尔之前转化。$EdabitState$ 中同样包含使用" 切片-选择" 方式生成 EdaBit 的方法。

这样，通过 $SpdzWiseFieldState$、$BeaverState$ 和 $EdabitState$ 三个状态，可以分别处理算术运算、布尔运算和秘密共享转化的关键信息，以支持安全多方计算中的不同操作和计算需求。这些状态的有效管理和更新可以提高计算的效率和安全性，确保正确执行安全多方计算协议中的各个步骤和运算。这些状态量和内核，始终存储在 SPU 的内核上下文 $KernelEvalContext$ 当中，任何拥有上下文变量的函数都可以访问、操作这些状态量和内核。

这里，我们给出一个宏观上数据流通的过程。用户想要计算整数 a 和布尔值 b 的乘法，在输入阶段，整数 a 会通过 IO 模块的 $toShares$ 函数变成秘密共享形式 $[\![a]\!]_{\mathbb{F}_p}$，即 $toShares(a) \rightarrow [\![a]\!]_{\mathbb{F}_p}$，同时布尔值 b 通过内核 $P2B$ 变成秘密共享形式 $[\![b]\!]_{\mathbb{F}_2}$，即 $P2B(b) \rightarrow [\![b]\!]_{\mathbb{F}_2}$。目前两个计算量所在的域不同，因此需要调用一次内核 $B2A$，将 $[\![b]\!]_{\mathbb{F}_2}$ 抬升到 $[\![b]\!]_{\mathbb{F}_p}$，最后调用内核 $MulAA$ 计算两者乘积，得到 $[\![ab]\!]_{\mathbb{F}_p} = MulAA([\![a]\!]_{\mathbb{F}_p}, [\![b]\!]_{\mathbb{F}_p})$。最终，通过内核 $fromShares$ 输出计算后的明文结果，计算过程结束。

秘密共享输入与重构 在 Spdzwise 协议中，秘密共享包含数据的半诚实秘密共享本身，以及数据消息验证码的半诚实秘密共享两部分。根据章节 3.1.2中介绍，秘密数据的输入除了包含一次半诚实秘密共享输入，以及一次与全局消息验证码密钥的半诚实秘密乘法两部分。SPU 当中提供了一种方便的、通过上下文动态调用任意内核的方法，使得该内核函数的设计非常简单，用公式表示即：

$$[\![x]\!] = P2ASH(x)$$
$$[\![mac_x]\!] = MulAASH([\![x]\!], [\![key]\!])$$

这里也能看出，在三方诚实大多数设定下，每个 Spdzwise 秘密共享所占空间为四个域上元素大小，其中两个用于存储数据本身的半诚实秘密共享，另外两个用于存储数据的消息验证码秘密共享。本项目中，秘密共享存储的顺序为：x_0, x_1, mac_0, mac_1。

具有同态性质的运算 根据 2.2.4所提到的内容，具有同态性质的运算（明文与密文的加法与乘法、密文与密文的加法）不需要参与方之间进行通信，即这些过程是非交互的，那么在这些过程中就不需要使用状态量 $Communicator$，各个参与方只需要关注自身运算即可，对于常数与秘密数的加法运算，需要额外调用一次常数与全局消息验证码密钥

的半诚实乘法。由于本文的算术运算均运行在 61-bit 梅森素数域，所以本工作额外实现了 61-bit 梅森素数域的实现。对于 $p = 2^{PE} - 1$ 的梅森素数域，包含如下运算性质：

- 模运算。计算 $x\%p$，可以降维到计算 $x' = x >> PE + x \wedge p$，此时 x' 可能比 $x\%p$ 多出一个 p，因此需要在判断 x' 是否比 p 大，如果更大则直接减去，就得到了取模的正确结果。这样，一次模运算就简化为了一次与运算、一次位运算、一次加法运算、一次比大小运算和一次减法运算。

- 加法运算。对于两个梅森素数域上的元素，其和必然在 $[0, 2p-1)$ 之间，也就是说若和大于 p，只需减去一个 p 即可，这样就把一次带模加法运算中的模运算简化为了一次比大小和一次减法运算。

- 乘法运算。梅森素数域上的乘法运算满足：$Mul(a, b) = Add(((a * b) >> PE), ((a * b) \wedge p))$，这样，一次带模乘法运算就降低到了一次乘法运算、一次移位运算、一次与运算和一次域上带模加法运算。

秘密数乘法运算 根据章节 3.1中所介绍，计算一次 SPDZ 秘密共享乘法，包含了两次半诚实安全协议的乘法运算。由于数据均存储在 $ArrayRef$ 中，所以每个操作都需要同时处理多组数据。而又因为同一批数据之间是相互无关的，所以在使用循环语句处理每个数据时，循环语句可以使用多线程并发加速。计算 $[\![x]\!]$ 与 $[\![y]\!]$ 的乘法运算，包含的两个半诚实协议乘法分别为 $[\![x]\!] \cdot [\![y]\!]$ 和 $[\![\Delta x]\!] \cdot [\![y]\!]$，而两者之间本身也互不影响，所以这两次乘法也可以压缩进一轮通信。另外工程中需要注意的一点，原始论文中考虑的最终检查是在整个协议所有运算执行结束之后进行，并且通过直接打开 mac 私钥 Δ 来存储，但是在实际工程中，或许存在着任务较大（例如训练机器学习模型）的场景，其乘法运算术量巨大，导致一次性存储所有数据与消息认证码的开销十分巨大，因此这里本项目实现中设置一个存储的上限，即每保存超过若干（例如 10^6）个待验证数据二元组，就进行一次检查，并且确保检查本身不能泄露 mac 私钥的任何信息，这也是为什么在检查阶段没有参考原始论文方案的原因。这样，在本工作的实现中，每次调用乘法指令都涉及到一个或者多个乘法，把这些乘法的结果以及其消息验证码存到 $SpdzWiseFieldState$ 当中，$State$ 会在保存后判断缓冲区大小是否已经超过阈值，若超过则会分批次对数据进行一致性校验。

至于矩阵运算部分，矩阵运算的本质是向量内积运算，而每一维度乘法之间的元素又是相互无关的，因此我们可以把矩阵乘法“展开”成很多单独的秘密数乘法运算，然后根据 SIMD 的优化思想，可以在一轮乘法运算后得出所有的结果。例如，对于如下矩阵：

$$\begin{pmatrix} [\![x_{00}]\!] & [\![x_{01}]\!] \\ [\![x_{10}]\!] & [\![x_{11}]\!] \end{pmatrix} * \begin{pmatrix} [\![y_{00}]\!] & [\![y_{01}]\!] \\ [\![y_{10}]\!] & [\![y_{11}]\!] \end{pmatrix}$$

可以提取出需要计算的数据，即

$$\vec{lhs} = (\llbracket x_{00} \rrbracket, \llbracket x_{01} \rrbracket, \llbracket x_{00} \rrbracket, \llbracket x_{01} \rrbracket, \llbracket x_{10} \rrbracket, \llbracket x_{11} \rrbracket, \llbracket x_{10} \rrbracket, \llbracket x_{11} \rrbracket)$$
$$\vec{rhs} = (\llbracket y_{00} \rrbracket, \llbracket y_{10} \rrbracket, \llbracket y_{01} \rrbracket, \llbracket y_{11} \rrbracket, \llbracket y_{00} \rrbracket, \llbracket y_{10} \rrbracket, \llbracket y_{01} \rrbracket, \llbracket y_{11} \rrbracket)$$

这样就可以一轮计算出所有所需的乘法运算。

“切片选择”生成 Beaver 三元组 前文详细介绍了如何使用“切片选择”方式生成 Beaver 三元组。在实现上，仍然有一些值得优化的点。首先，每次生成 Beaver 三元组的大小为 $batchsize$，而实际可能并不会使用到如此多的三元组，因此需要对多余的三元组进行存储，以后续使用。另外有一个更加有意义的优化，借助 SIMP 的思想，在生成 Beaver 三元组过程中所涉及的操作均为比特层面，本项目实现中思考把 64 个比特压缩为一个 $uint64\_t$ 数据，所有运算操作均不变，这就相当于一个指令同时处理了 64 组数据，虽然不改变通信量，但是能够显著减少通信轮数。况且考虑到计算机指令本身支持 64 比特数据的直接运算，因此执行 64 次布尔类型数据的逻辑运算时间要高于执行一次 64 比特数据的逻辑运算。例如，现在设参数 $batchsize = 20000, bucketsize = 4$，那么每次切片选择能够生成 $20,000 * 64 = 1,280,000$ 个二进制 Beaver 三元组，即使第一次请求获取 $1,000,000$ 个三元组，由于切片选择方法生产出的三元组数量完全取决于 $batchsize$，因此仍然会生成 $1,280,000$ 个三元组，多余的 $280,000$ 个三元组将会缓存起来。若下次请求 $1,500,000$ 个三元组，如果没有缓存机制，程序将会运行两次切片选择方法，生成 $2,560,000$ 个三元组并丢弃大量，但是由于缓存机制，本次只需要额外生成 $1,220,000$ 个三元组，一次切片选择即可完成，并且可以结余 $60,000$ 个三元组缓存起来。另外，虽然 SPU 当中布尔秘密共享采用压位存储，即存储所使用的数据类型若为 uint64，那么一个变量会最多存储 64 个信息，况且在切片选择生成三元组的时候也使用了这种压位存储思想，看起来两者是天然的吻合，可以直接将切片选择的输出应用到二进制运算，但是本工作中并没有这么做，因为很多时候压位存储并不会存满 64 比特，大部分的空间是没有使用的，在这种情况下如果直接分配 64 个二进制 beaver 三元组，那么会浪费掉很多可信三元组。因此本工作将切片选择方法的输出进行比特重排，以满足使用需求，减少浪费。该过程可以用代码 4-1表示。

代码 4-1 切片选择的输出重构

```
size_t size_bit = sizeof(OutT) * 8;
OutT mask = (1 << size_bit) - 1;

ArrayRef out(makeType<spdzwisefield::BinTripleTy>(out_type), size);
auto out_view = ArrayView<std::array<std::array<OutT, 2>, 3>>(out);

pforeach(0, size, [&](uint64_t idx) {
  int row = idx / compress;
```

```
  int col = idx % compress;
  auto& [a, b, c] = out_view[idx];
  const auto [x, y, z] = trusted_triples_bin_->at(row);

  a[0] = (x[0] >> (size_bit * col)) & mask;
  a[1] = (x[1] >> (size_bit * col)) & mask;
  b[0] = (y[0] >> (size_bit * col)) & mask;
  b[1] = (y[1] >> (size_bit * col)) & mask;
  c[0] = (z[0] >> (size_bit * col)) & mask;
  c[1] = (z[1] >> (size_bit * col)) & mask;
});

trusted_triples_bin_->erase(trusted_triples_bin_->begin(),
                            trusted_triples_bin_->begin() + need);

return out;
```

“切片选择”生成 EdaBits 这里仍然使用切片选择的思想生成 EdaBits。根据前文描述，整个过程分为“生成随机 EdaBit”以及“切片选择筛选”两个过程，并且两个过程均存在布尔电路计算，加上实际应用 EdaBits 进行秘密共享转化过程也存在二进制电路模拟计算，因此在实现该功能之前，本工作先实现了两种电路：全加器电路，二进制补码电路。并且为了让两种电路能够可选半诚实安全性与恶意安全性，将电路中需要的与门计算单独抽象出来，根据不同的安全需求来决定选择使用半诚实安全性二进制与门运算或恶意安全性二进制与门运算。虽然本工作默认所有算术计算所在域为 61 比特梅森素数域，此处生成的 EdaBit 也是 61 比特下，但实际上这套方法以及本工作的实现适用于任意小于等于 128 的长度（本工作不考虑高精度整数），因此若后面若有 127 比特梅森素数域上计算的需求，这套代码仍然可以使用。同样的，EdaBit 的生成也存在缓存机制，减少浪费。

ABY3[7] 方法生成 TruncPair 根据前文描述，本文使用 ABY3 当中的算术截位运算方法，其方法中以布尔电路计算为主，这恰好使得生成 EdaBit 过程中的电路计算函数得到更大程度的利用。目前此部分的实现是按需生成，即根据每次的需求量来生成对应数量，这是因为这种生成 Truncation Pair 的方法没有每批次生成数量的限制以保证安全性（这与切片选择方法不同，切片选择方法的安全性与批次大小有关，见 3.1）。然而目前的实现方式仍然存在效率低下的问题，因为生成 Truncation Pair 的通信轮数为常数轮，为了能够降低通信开销，每批次应尽量多生成，但根据测试结果发现，实际计算当中调用该模块频繁，每次请求量小，这导致通信轮数开销巨大，为了得到更好的优化，仍然应该使用上文中关于缓冲的优化，每批次生成较多（如 20,000 个），然后将多余数对存入缓冲区。本部分的实现也具有一个新特点：前面提到的计算过程均为对称过程，也就是

每个参与方所需要进行的计算、交互是一致的，对称过程下编写代码较为整体、统一，而非对称过程则需要考虑不同编号的参与方所做的内容，并且在交互过程应分别考虑每个参与方的行为（例如有些参与方需要发送数组 $prev$ 的内容，有些需要发送 $next$ 的内容）。

C/S 模型的 IO 模块 前文提到，在 C/S 模型下，需要用户对输入进行分片操作，这个操作与复制秘密共享的输入是同样的形式。具体来讲，本工作设计的过程中，需要将秘密数 x 拆分成 x_1, x_2, x_3 使得 $x_1 + x_2 + x_3 = x$，然后分组为 $(x_1, x_2), (x_2, x_3), (x_3, x_1)$ 分别发送给各个计算方。这个操作是在域上进行。计算过程如下：

- **输入:** 客户端输入一个域上元素 $x \in \mathbb{F}_p$ 与分片数量 N。
- **执行过程:**
  1. 创建空数组 $split$，设置大小为 N，并令 $split[0] = x$。
  2. 从下标 $i = 1 \ldots N-1$，将 $split[i]$ 设置为随机元素，并将 $split[0]$ 减去该元素。
  3. 此时，可以满足 $\sum_{N-1}^{i=0} split[i] = x$。
- **输出:** 返回 $split$ 数组。

协议的端到端执行 隐语 SPU 框架的核心特点是用户友好与高性能兼顾，除去底层使用 C++ 开发之外，其上层用户使用时可以直接编写 Python 代码。在 SPU 当中，创建一个完备的协议，除了要实现上述需要的 API 与 IO 模块之外，还需要将协议注册到 spu.proto 当中。该文件通过 enum 类型定义了不同协议的编号，注册新的协议，需要将协议名称和编号一并写入该文件。本项目中定义 $SPDZWISEFIELD = 5$（前面四个分别为 Ref2K[8], Semi2K, ABY3[7], Cheetah[20]）。随后，需要在 SPU 的工厂函数中绑定协议对应的 $Object$ 构造函数，以让上层明白协议的本体在什么位置，该构造函数返回一个 $Object$ 类型，而这个类型包含了协议的所有内容，包括内核、状态等。此后，便可以编写 Python 程序，通过配置文件指定协议名称、协议所运行的环或域、协议参与方信息等，来完成 Python 层面的开发工作。

第五章　实验结果

本章节会展示本项目实现的协议同 ABY3、明文计算运行训练线性回归机器学习模型的效率对比，同时还会对比本项目实现的” 切片选择” 方案生成 Beaver 三元组、Edabit 方法与 MP-SPDZ[4] 中的方案效率进行对比。代码实现基于本文写作时 SPU 的最新版本，最后一次 commit 编号为 8100491d9b9ecfdbccca8abb8b94e4e3f3ed33f0。所有代码均可在https://github.com/DuanYuFi/spu/tree/7257b0d07e7ebd3218850e75443df9d7380c81bc找到，所有 MP-SPDZ[4] 框架上的测试代码均可在https://github.com/DuanYuFi/MP-SPDZ-test/tree/ff13e9f5052e603780f114ce6674da46647f183d找到。

5.1　实验环境

所有实验中，本项目的协议均运行在 61-bit 梅森素数域上，其中 $p = 2^{61} - 1$，ABY^3 协议均运行在 $Z_{2^{64}}$ 环上。实验中所有运算术据均小于 2^{20}，以达到 40-bit 统计安全强度。

实验均在本地运行，即在同一台机器中模拟三个参与方进行运算，因此不考虑网络设置。本实验使用腾讯云 S5.2XLARGE16 实例设置，配备 8 核 Intel(R) Xeon(R) Platinum 8255C CPU @ 2.50GHz 处理器和 16GB 内存。基准测试中，本实验限制平均每个参与方分配 1 个核，即限制总计算资源为 3 核，在机器学习训练测试下，不对计算资源做限制，即整体拥有 8 核算力。为了确保 SPU 框架在基准测试中没有使用更多的算力，（考虑到本工作中，框架的线程并发只出现在 pforeach 函数）将 pforeach 限制在仅使用 1 个线程的算力；为了确保 MP-SPDZ 框架在基准测试中没有使用更多的算力，使用了 Linux $taskset$ 指令限制了核数，并且在运行的全过程中使用 $htop$ 指令随时观察 CPU 的负载情况。

5.2　基准测试

生成 Beaver 三元组基准测试　本项目首先对在 SPU 当中实现的 FLNW17” 切片选择” 方案进行测，对比对象是目前已知的最优框架 MP-SPDZ 中所实现的 FLNW17” 切片选择” 方案。测试中，本项目对比了两者在三个 CPU 算力限制下的性能。对于 SPU，本项目实现了两个不同的版本来进行测试，区别在于是否采用了3.2.2提到的有关哈希的优化，使用 Google Test 框架编写单元测试，调用一次”gen_bin_triples” 函数，参数选择为 $batchsize = 20,000, bucketsize = 4$，在切片选择方法中选择使用 32 位、64 位、128 位压缩（即存储类型设置为 uint32、uint64、uint128），每次均生成 $1,000,000$ 个对应存储类型的三元组，这样最终生成的三元组数量就是 $32M, 64M, 128M$。MP-SPDZ 中，使用框架中自带的 Preprocessing 类型中”get_triple” 的方法，每次获取 32/64/128 比特的三元组，不断调用 $1,000,000$ 次，这种方法看起来效率低下，但是考虑到本工作的实验均只考虑每个参与方一核算力，因此此处不考虑并行的优化。从表 5-1 可以看出，在通信优化下，在 SPU 中实现的方案通信量能够与 MP-SPDZ 几乎一致，同时也是论文 FLNW17[11] 当

协议	生成三元组数量	时间（毫秒）	每个参与方通信量（MB）
非通信优化 SPU	32M	605	49.5974
	64M	979	99.1947
	128M	2343	198.389
通信优化 SPU	32M	926	39.6762
	64M	1723	76.3003
	128M	3679	152.601
MP-SPDZ	32M	740	38.1521
	64M	1489	76.3041
	128M	3033	152.608

表 5-1 在设置 batch-size=20,000，bucket-size=4 下，生成 32M、64M、128M 数量三元组的耗时（毫秒）和各参与方通信量（MB）

协议	生成 EdaBit 数量	时间（毫秒）	每个参与方通信量（MB）
SPU	20,000	1391	-
	100,000	6459	-
	200,000	12680	-
MP-SPDZ	20,000	2124	55.6891
	100,000	7234	195.742
	200,000	11670	317.745

表 5-2 在设置 batch-size=20,000，bucket-size=4 下，生成 20,000、100,000、200,000 数量 EdaBits 的耗时（毫秒）和各参与方通信量（MB）

中的理论最优。在非通信优化下，在 SPU 中的耗时相比 MP-SPDZ 约有 18% 到 34% 的提升。

生成 EdaBit 基准测试 本项目其次对 SPU 当中实现的基于切片-选择方法生成 Edabit 的方法[15]，同 MP-SPDZ 框架进行对比。由于 SPU 当中的通信模块采用异步发送，因此对于通信轮数不好界定。而生成 EdaBit 的实现中采用了基于三方不经意传输的 ABY^3[7] 比特转化方案（ABY^3 的 5.3 章节，段落 Bit Injection），这种方案在 SPU 原始实现中是通过异步发送数据，因此不好界定通信轮数，所以 SPU 中无法统计这方面的通信量。在 SPU 中，本文仍然使用 Google Test 框架，选择参数为 $batchsize = 20,000, bucketsize = 4$，生成的 EdaBit 大小为 61bit，调用 $EdabitState$ 中的"gen_edabit" 函数每次生成 $20,000$、$100,000$、$200,000$ 个 EdaBit。在 MP-SPDZ 框架中，本文使用了原始论文作者推荐的基准测试代码[1]，将其更改为每个参与方单线程并修改对应参数测试。根据表 5-2 的结果可以看出，在数据量较小的情况下（#$\leq 100,000$），SPU 的效率要优于 MP-SPDZ，而数据量增大后 SPU 相较于 MP-SPDZ 更慢。

[1] 见https://github.com/data61/MP-SPDZ/issues/998

协议	参数	时间（秒）	误差 *
Spdzwise-FLNW17	10pts, 200epochs	46.4265	0.7062217
	100pts, 20epochs	5.164111	2.731065
ABY^3	10pts, 200epochs	9.3563	0.706224
	100pts, 20epochs	1.7389	2.7310767
CPU	10pts, 200epochs	1.51537	0.7060817
	100pts, 20epochs	1.1173	2.7309072

表 5-3 在不同点数和不同迭代轮数下，训练线性回归模型的耗时（秒）和误差
* 此处误差为数据点与评估点距离平方的均值

5.3 机器学习模型训练

为了将协议实际应用在机器学习场景，本项目测试了分别在明文、ABY^3 协议[7] 和 Spdzwise Field 协议[9]+FLNW17[11] 上训练线性回归机器学习模型的效率。其中两部分切片-选择均采用 batch-size 为 20,000，bucket-size 为 4。线性回归模型中，本项目分别选用 10 个点、0.01 学习率、200 轮迭代以及 100 个点、0.01 学习率、20 轮迭代两种情况，并且采用数据点与评估点距离平方的均值来表示误差，即

$$error = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - y_i)^2$$

其中 (x_i, y_i) 为第 i 个数据点。该误差越小说明拟合越精确。实现中，由于本工作实现的协议接口完备，因此可以直接进行端到端的测试，使用 Python 编写测试代码。测试代码中，首先从两方 Alice 和 Bob 获取各自的数据，其中 Alice 提供点的横坐标，Bob 提供各点的纵坐标，分别设置参数 10 个点、200 轮迭代和 100 个点、20 轮迭代，使用 SPU 设备进行密态线性回归计算。对于密文计算，采用了 ABY^3 协议和本文设计的协议，对于明文运算，直接使用 CPU 和 Python 常规代码进行计算。根据表 5-3 的结果来看，密态计算与明文计算在相同设定下拥有相当的精确度。在计算时间上来看，恶意安全性协议的执行效率比半诚实安全性协议慢 5 倍，在不同参数下的差距都在常数级别，而密态计算与明文计算的效率难以比较，波动较大。除此之外，对比相同协议的不同参数设定，可以看出即使增加了点的数量，降低迭代次数也会使得密态计算时间大大缩短，Spdzwise+FLNW17 协议和 ABY^3 协议分别提升了 9 倍和 5.4 倍，相比之下明文运算仅提升 1.36 倍。

第六章　总结与展望

本毕设项目深入研究了诚实大多数条件下的恶意安全性三方隐私计算协议，并且将这种混合协议实现在了隐语 SPU[1] 框架中。本项目的实现是 SPU 框架中第一个全过程恶意安全性的协议，并且在框架中新增了 Cut-and-Choose[14]、EdaBit[15] 等基础设施，这为 SPU 框架提供了大量的基础设施，为后续开发更多恶意安全性协议做了铺垫。同时，本项目使用 SPU 进行了机器学习层面的性能测试，并得到了多种实验数据，这些实验数据既证明了 SPU 框架功能的完备性，与 MP-SPDZ 框架的对比也能看出该框架性能之高，这不仅满足了工业界的使用需求，还表明了 SPU 适合用于学术研究以获取研究者所创造协议的实验数据。

本项目后续还有很大的改进空间。在章节 3.1.5中，提到的生成 Truncation Pair 方案实际上可以优化为批次生成，因为每次调用均会产生一次通信轮数。在这里可以设定一个 batch size，如果需求超过这个值，那么直接一次性生成这些数量的 Truncation Pair，否则，生成 batch size 个 Truncation Pair，并将多余的保存以供后续使用。除此之外，目前协议中秘密共享转化的方案仍然是基于统计安全性的，并且所支持数据范围很小，后续可以将这部分提升为信息论安全性，这样就能够使用更大范围的数据。最后，代码中仍然存在可以优化通信和计算的地方，通过对这部分内容的优化，希望最终能够超越 MP-SPDZ 的性能，并且能够使用恶意安全性隐私保护协议训练大型模型如 GPT 等[1]。

[1]此处灵感来自于 SPU 已经实现的功能，见https://github.com/secretflow/spu/tree/main/examples/python/ml/flax_gpt2

参考文献

[1] AntGroup. SecretFlow: A unified framework for privacy-preserving data analysis and machine learning. Version 0.7.11, https://www.secretflow.org.cn/.

[2] Ito Mitsuru, Saito Akira, Nishizeki Takao. Secret sharing scheme realizing general access structure [J]. Electronics and Communications in Japan (Part III: Fundamental Electronic Science). 72 (9). 1989: 56–64.

[3] Hazay Carmit, Lindell Yehuda. A Note on the Relation between the Definitions of Security for Semi-Honest and Malicious Adversaries. [J]. IACR Cryptology ePrint Archive. 2010. 2010, 10: 551.

[4] Keller Marcel. MP-SPDZ: A Versatile Framework for Multi-Party Computation [C/OL]. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. 2020 .

[5] Liu Chang, Wang Xiao Shaun, Nayak Kartik et al. ObliVM: A Programming Framework for Secure Computation [C]. In 2015 IEEE Symposium on Security and Privacy. 2015 : 359–376.

[6] Knott B, Venkataraman S, Hannun AY et al. CrypTen: Secure Multi-Party Computation Meets Machine Learning [C]. In arXiv 2109.00984. 2021 .

[7] Mohassel Payman, Rindal Peter. ABY^3: A Mixed Protocol Framework for Machine Learning [C/OL]. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. 2018 : 35–52.

[8] Cramer Ronald, Damgård Ivan, Escudero Daniel et al. SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part II [M]. 2018: 769–798.

[9] Chida Koji, Genkin Daniel, Hamada Koki et al. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries [C]. In Advances in Cryptology – CRYPTO 2018. Cham. 2018 : 34–64.

[10] Damgård Ivan, Pastro Valerio, Smart Nigel et al. Multiparty Computation from Somewhat Homomorphic Encryption [C]. In Advances in Cryptology – CRYPTO 2012. Berlin, Heidelberg. 2012 : 643–662.

[11] Furukawa Jun, Lindell Yehuda, Nof Ariel et al. High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority [C]. In Advances in Cryptology – EUROCRYPT 2017. Cham. 2017 : 225–255.

[12] Araki Toshinori, Barak Assi, Furukawa Jun et al. Optimized Honest-Majority MPC for Malicious Adversaries —Breaking the 1 Billion-Gate Per Second Barrier [C]. In 2017 IEEE Symposium on Security and Privacy (SP). 2017 : 843–862.

[13] Beaver Donald. Efficient Multiparty Protocols Using Circuit Randomization [C]. In Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings. 1991 : 420–432.

[14] Chaum David. Blind signature system [C]. In Advances in cryptology. 1984 : 153–153.

[15] Escudero Daniel, Ghosh Satrajit, Keller Marcel et al. Improved primitives for MPC over mixed arithmetic-binary circuits [C]. In Annual International Cryptology conference. 2020 : 823–852.

[16] Rotaru Dragos, Wood Tim. MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security [C]. In Progress in Cryptology – INDOCRYPT 2019. Cham. 2019 : 227–249.

[17] Araki Toshinori, Furukawa Jun, Lindell Yehuda et al. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority [C/OL]. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA. 2016 : 805–817.

[18] Yao Andrew C. Protocols for secure computations [C]. In 23rd annual symposium on foundations of computer science (sfcs 1982). 1982 : 160–164.

[19] Wagh Sameer, Tople Shruti, Benhamouda Fabrice et al. Falcon: Honest-Majority Maliciously Secure Framework for Private Deep Learning [C/OL]. In Privacy Enhancing Technologies Symposium (PETS). 2021 .

[20] Huang Zhicong, jie Lu Wen, Hong Cheng et al. Cheetah: Lean and Fast Secure Two-Party Deep Neural Network Inference [C/OL]. In 31st USENIX Security Symposium (USENIX Security 22). Boston, MA. 2022 : 809–826.

致　　谢

我要感谢我的指导老师毕经国老师，他是带领我入门密码学的老师，在我的本科学习期间，他不仅给予了我专业知识方面的指导，还在思考和研究方法上给予了我许多启示和帮助，让我对密码学产生了浓厚的兴趣。同时，我要感谢我未来的博士生导师张超教授，在我的本科学习期间，我有幸能够加入他的研究团队，跟随他的指导进行隐私计算方向的研究学习。张教授在我进行研究的过程中给予了我极大的支持和指导，让我深入掌握了密码学的知识和学术研究的规范和方法。我将继续跟随他攻读博士学位，并深切感谢他对我未来学术生涯的支持和鼓励。

其次，我要感谢我的父母，感谢他们对我的养育之恩和无私的付出。他们一直默默地支持我，不论是经济上还是精神上，他们的爱和支持一直都是我前进的动力。我要感谢我的姐姐，她不仅是我的亲人，更是我的知心朋友。在我学习和生活中遇到困难和挫折时，她总是耐心地倾听我倾诉，给予我最温暖的鼓励和支持。我还要感谢我的女友。在我学习和生活中，她一直是我的坚实后盾。在我面对压力和困难的时候，她给予了我最温暖的陪伴和关怀，帮助我度过了难关。我将永远珍视他们的爱和关怀，以更加优秀的成绩回报他们。

最后，感谢一路走来的朋友们，感谢初中以来的挚友宿阳阳，感谢我的室友以及206 宿舍同学，他们在生活上增添了欢乐，让我感受到了家一般的温馨和舒适，我们一起分享了快乐和烦恼，度过了难忘的时光，这些回忆将永远珍藏在我心中。感谢赵哥 Retr0 带领我踏入 CTF 竞赛，感谢刘佬 LordRiot 和薛哥 NaIrW 在我密码学学习中提供的帮助，感谢宋一凡学长、哈博在我学习隐私计算过程中给予的支持。

为天地立心，为生民立命，为往圣继绝学，为万世开太平。

本研究项目得到蚂蚁集团资助。Sponsored by Ant Group.