《数据安全》实验报告

姓名：管昀玫 学号：2013750 班级：计科一班

**实验名称：**

零知识证明：以零知识证明方式提供财富达标证明

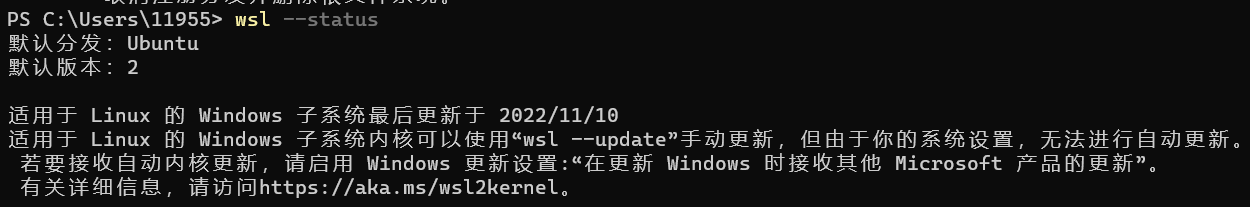
**实验要求：**

1. 给出libsnark开发环境安装（使用ubuntu或者windows搭建）
2. 给出非交互零知识证明的基本知识描述
3. 给出“6.2 开发框架 libsnark”的详细介绍
4. 基于 6.3 应用案例∶以零知识证明方式提供财富达标证明，完成具体开发

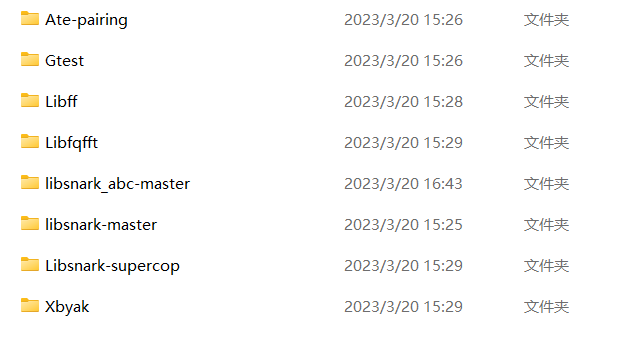
**实验过程：**

1. libsnark开发环境安装

我选择的环境是wsl2



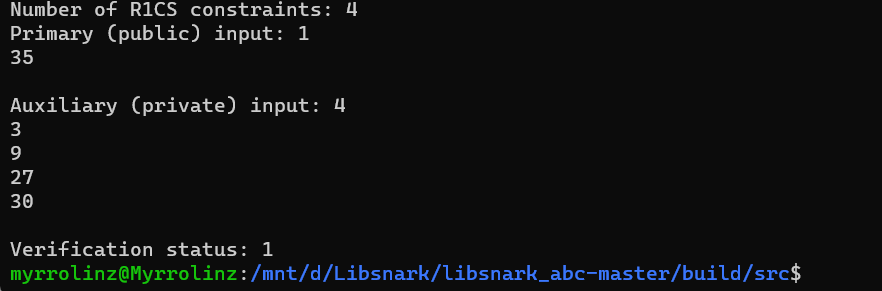
安装过程较为繁琐。由于我安装libsnark是在老师发讲解视频之前，并不知道在实验报告内要体现开发环境安装过程，因此我没有安装的截图。但是我尽可能说明安装过程和踩过的坑；

* 首先创建Libsnark文件夹，并依次下载libsnark\_abc-maser、libsnark、以及六个子模块并解压、重命名，暂时以平行结构存储这8个文件夹以便之后的移动。  
  
* 根据对应的系统（我用的是wsl2，对应为Ubuntu 22.04 LTS）执行以下命令：

|  |
| --- |
| sudo apt install build-essential cmake git libgmp3-dev libprocps-dev python3-mar  kdown libboost-program-options-dev libssl-dev python3 pkg-config |

由于我之前已经装过build-essential cmake git等常用的依赖项，因此只需要装后几个依赖项即可。

* 安装子模块xbyak：将下载的子模块xbyak的内容复制粘贴到/Libsnark/libsnark\_abc-master/depends/libsnark/depends/xbyak，并执行sudo make install命令
* 安装子模块ate-pairing：同上，将下载得到的文件夹 Ate-pairing 内的文件复复制~/Libsnark/libsnark\_abc-master/depends/libsnark/depends/ate-pairing中，并执行make -j和test/bn指令
* 安装子模块libsnark-supercop：同上，将下载得到的文件夹 Libsnark-supercop 内的文件复制到~/Libsnark/libsnark\_abc-master/depends/libsnark/depends/libsnark-supercop中，并执行指令./do
* 安装子模块：将下载得到的文件夹 Gtest 内的文件复制到~/Libsnark/libsnark\_abc-master/depends/libsnark/depends/gtest即可  
  踩坑：第一次安装的时候由于gtest的操作较为简单我就忘了复制，导致libsnark编译安装的时候出现失败，后来才排查出来是gtest忘了复制。复制并编译的过程一定要细心且耐心。
* 安装子模块libff：同上，Libff 内的文件复制到~/Libsnark/libsnark\_abc-master/depends/libsnark/depends/libff。但是要注意，libff中的的depends内还有ate-pairing和xbyak，它们也是空的，要注意把内容复制进去。并执行指令：mkdir build | cd build | cmake .. | make | sudo make install | make check
* 安装子模块：libfqfft：同上，将文件夹 Libfqfft 内的文件复制到~/Libsnark/libsnark\_abcmaster/depends/libsnark/depends/libfqfft内。要注意depends中的ate-pairing、gtest、libff、xbyak都是空的，注意复制。之后执行的指令和上面libff相同
* Libsnark编译安装：在~/Libsnark/libsnark\_abc-master/depends/libsnark执行指令：mkdir build | cd build | cmake .. | make | make check
* 整体编译安装：在~/Libsnark/libsnark\_abc-master中执行以下指令：mkdir build | cd build | cmake .. | make
* 运行代码：在src中执行命令：./test，出现以下界面：



说明zkSNARK环境安装成功，并跑通了demo。

1. 给出非交互零知识证明的基本知识描述

非交互零知识证明（Non-interactive zero-knowledge proof，NIZK）是一种密码学概念，用于证明某些信息的真实性，而不需要交互式的通信过程。在非交互零知识证明中，证明者**只需要向验证者发送一条消息**，并在不泄漏任何有关证明的信息的情况下，使验证者相信该消息是真实的。

非交互零知识证明可以用于许多应用，例如电子投票、数字签名、密码学协议等。它们的主要好处是，它们允许在不暴露任何有关被证明信息的详细信息的情况下，证明信息的真实性。

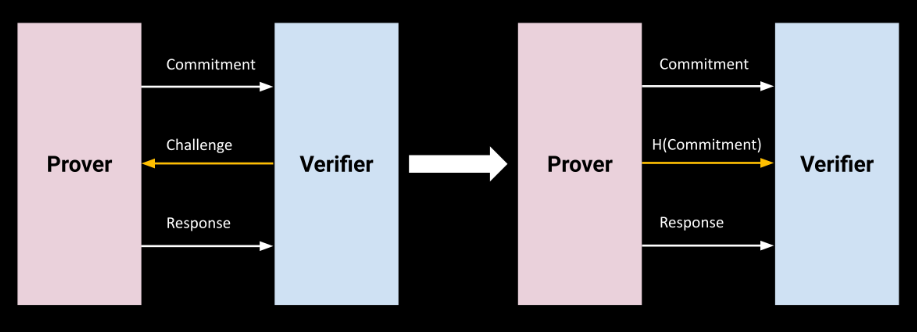
非交互零知识证明是建立在复杂性理论的基础上的，其中涉及到难以计算的问题。它们通常基于密码学哈希函数、数字签名、零知识证明以及其他密码学工具。

将交互式零知识证明直接转为非交互零知识证明会面临一些挑战，即减少随机挑战和回应挑战两个步骤后，如何防止证明方伪造证明成为设计的关键。

目前主流的非交互式零知识证明的构造方法有两种，一是基于随机预言机并利用Fiat-Shamir启发式实现，二是基于公共参考字符串CRS (Common Reference String)模型实现。

1. **Fiat-Shamir 启发式（Heuristic）**

其基本过程如下图所示：



1. 系统初始化：首先，证明者和验证者需要共同选择一个安全参数n，以及一些固定的公共参数，如生成元g和模数p。这些参数可以在系统初始化时生成，并且在协议的整个过程中都不会改变。
2. 证明生成：证明者选择一个私有的随机数r，并计算出一个公开的值t，使得t = g^r mod p。然后，证明者将t发送给验证者。
3. 挑战生成：验证者选择一个随机的挑战c，并将其发送给证明者。
4. 响应生成：证明者根据挑战c计算出一个响应值s，使得s = r + cx mod n，并将其发送给验证者。
5. 验证：验证者接收到证明者发送的t和s后，根据公共参数g和p，以及证明者的响应s和挑战c来计算出一个值t'，使得t' = g^s \* t^c mod p。然后，验证者将计算出的t'与证明者发送的t进行比较，如果它们相等，则验证成功，否则验证失败。

在上述过程中，证明者仅需要向验证者发送一个值t和一个值s，而不需要进行交互式的证明过程。证明者通过使用随机数r来引入随机性，并根据挑战c计算响应s来证明自己拥有私有的信息，从而达到身份验证的目的。同时，由于证明者仅仅发送了一个固定的值t和一个响应s，因此该协议具有零知识证明的特性，即不泄露任何关于证明者私有信息的信息。

**非交互式Schnorr协议**就是基于随机预言机并利用Fiat-Shamir变换实现的非交互式零知识证明协议。Fiat-Shamir变换，又叫Fiat-Shamir Heurisitc（启发式），或者Fiat-Shamir Paradigm（范式），由Fiat和Shamir在1986年提出，其特点是可以将交互式零知识证明转换为非交互式零知识证明，思路就是用公开的哈希函数的输出代替随机的挑战。

（1）承诺阶段：Alice均匀随机选择r，并依次计算R=r\*G, c=Hash(R,PK), z=r+c\*sk，然后生成证明(R,z)。

（2）验证阶段：Bob(或者任意一个验证者)计算c=Hash(PK,R)，验证z\*G?==R+c\*PK。

为了不让Alice进行造假，在交互式Schnorr协议中需要Bob发送一个c值，并将c值构造进公式中。所以，在非交互式Schnorr协议中，如果Alice选择一个无法造假并且大家公认的c值并将其构造进公式中，问题就解决了。生成这个公认无法造假的c的方法是使用随机数预言机。

1. **公共参考字符串（CRS）**

CRS是一种随机字符串，由信任的第三方生成并分发给证明者和验证者。证明者使用这个字符串来生成证明，而验证者使用它来验证证明。CRS应该满足特定的性质，如随机性、唯一性和不可预测性。

基于CRS的非交互式零知识证明的**步骤**：证明者使用CRS来生成证明，并将证明发送给验证者。验证者使用CRS来验证证明的真实性，而不需要与证明者进行任何交互。

基于CRS的非交互式零知识证明的**优点**：由于不需要进行交互，这种证明技术可以更加高效地完成证明过程。此外，使用CRS可以减少通信和计算成本。

基于CRS的非交互式零知识证明的**应用**：这种证明技术在许多领域都有广泛的应用，包括密码学、区块链、数字身份验证和云计算等。

zkSNARK(zero-knowledge Succinct Non-interactive Arguments of Knowledge)就是一类基于公共参考字符串CRS模型实现的典型的非交互式零知识证明技术。zk-SNARKs是一种用于零知识证明的框架，它可以在不暴露任何私人信息的情况下验证某些事实的真实性。zk-SNARKs的全称是"零知识可扩展的非交互式证明"，其中zk代表"零知识"，SNARK代表"可扩展的非交互式证明"。zkSNARK中比较典型的协议有Groth10、GGPR13、Pinocchio、GRoth6、GKMMM18等。

它是一种基于双线性对（bilinear pairings）的非交互式零知识证明系统。zk-SNARKs能够在证明的同时，将证明的大小大幅缩小，并且具有高度的隐私保护性。

它有以下的技术特征：

• 简洁性：最终生成的证明具有简洁性，也就是说最终生成的证明足够小，并且与计算量大小无关。

• 无交互：没有或者只有很少的交互。对于zkSNARK来说，证明者向验证者发送一条信息之后几乎没有交互。此外，zkSNARK还常常拥有“公共验证者”的属性，意思是在没有再次交互的情况下任何人都可以验证。

• 可靠性：证明者在不知道见证（Witness，私密的数据，只有证明者知道）的情况下，构造出证明是不可能的（这样的证明系统叫作一个Argument）。

• 零知识：验证者无法获取证明者的任何隐私信息。

它的证明过程可以分为以下几步：

1. 前置条件：系统需要预设一些安全参数和公共参数，如椭圆曲线、一个大质数等。
2. 确定语句和目标：假设有一个语句S，要证明其为真。同时，有一个目标T，证明者需要证明自己拥有某些信息，以便进行S的证明。
3. 构造电路：证明者需要将语句S表示为一种电路，该电路包括多个逻辑门和变量。证明者需要将电路的结构和布尔运算关系确定下来，然后将其转换成一个有向无环图（DAG）。
4. 计算约束：对于电路中的每个门和变量，证明者需要计算出一组约束条件，这些约束条件需要满足电路的结构和布尔运算关系。对于一个门，其约束条件是一个等式，对于一个变量，其约束条件是一个限制。
5. 构造公共参数：根据上述约束条件和公共参数，构造一组验证密钥，并向验证者公开。
6. 计算证明：证明者需要选择一个随机的密钥，然后将其用于对电路进行计算。计算的过程中，证明者需要根据密钥和约束条件计算出每个变量的值，并将其作为证明。
7. 生成证明：根据上述计算结果，证明者需要生成一个证明，该证明包括两部分：一个证明密文和一个验证密钥。证明密文包括电路的输入、输出和每个门的计算结果，验证密钥用于验证证明的有效性。
8. 验证证明：验证者需要使用公共参数和验证密钥来验证证明的有效性。验证者需要检查电路的输入和输出是否满足约束条件，以及每个门的计算结果是否正确。如果证明有效，则验证者可以确定语句S为真，否则证明无效。

zk-SNARKs的应用广泛，可以用于加密货币、区块链、身份验证和隐私保护等领域。

3. 给出“6.2 开发框架 libsnark”的详细介绍

**Libsnark开发框架**是一个开源的C++代码库，旨在帮助开发者构建零知识证明（Zero-Knowledge Proofs）应用程序，其中zkSNARK是一种广泛使用的零知识证明协议之一，目前已在GitHub上开源。

libsnark框架提供了许多用于构建zkSNARK协议的基本构建块和工具，例如：

* 可插拔的Hash函数和加密算法
* 各种曲线上的抽象代数（如有限域，椭圆曲线）
* 生成与验证zkSNARK证明所需的算法
* 用于编写zkSNARK程序的高级语言库

libsnark框架提供了多个通用证明系统的实现，其中使用较多的是BCTV14a和Groth16。

Groth16计算分成3个部分。

- Setup∶针对电路生成证明密钥和验证密钥。

- Prove∶在给定见证（Witness）和声明（Statement）的情况下生成证明。

- Verify∶通过验证密钥验证证明是否正确。

查看libsnark/libsnark/zk\_proof\_systems路径，就能发现libsnark对各种证明系统的具体实现，并且均按不同类别进行了分类，还附上了实现依照的具体论文。其中：

• zk\_proof\_systems/ppzksnark/r1cs\_ppzksnark对应的是BCTV14a

• zk\_proof\_systems/ppzksnark/r1cs\_gg\_ppzksnark对应的是Groth16

在Groth16中，ppzksnark是指preprocessing zkSNARK。这里的preprocessing是指可信设置(trusted setup)，即在证明生成和验证之前，需要通过一个生成算法来创建相关的公共参数（证明密钥和验证密钥），这个提前生成的参数就是公共参考串CRS。

**可复用电路Gadget**：gadgetlibl和gadgetlib2是实现常用的算术电路预制到库中的工具包，其中包含了一些基本运算的R1CS，比如 sha256在内的哈希计算、默克尔树、pairing 等电路实现。

comparison\_gadget(

                    protoboard<FieldT>& pb,

                    const size\_t n,

                    const pb\_linear\_combination< FieldT > &A,

                    const pb\_linear\_combination< FieldT > &B,

                    const pb\_variable<FieldT> &less,

                    const pb\_variable<FieldT> &less\_or\_eq,

                    const std::string &annotation\_prefix= ""

)

该Gadget需要传入参数较多：n表示参与比较的数的比特位数，A和B分别为需要比较的两个数，less和less\_or\_eq用来标记两个数的关系是“小于”还是“小于或等于”。其实现原理简单来说是把A和B的比较，转化为按位表示。

如下代码创建了相关变量，并将A和B与原型板相连，把B的值设为88，代表数值上限。

protoboard<FieldT>pb;

pb\_variable<FieldT>A,B,less,less\_or\_eq;

A.allocate(pb,"A");

B.allocate(pb,"B");

pb.val(B)=88;

less.allocate(pb,"less");

less\_or\_eq.allocate(pb,"less\_or\_eq");

使用comparison\_gadget创建cmp，并把前面的参数传入，调用Gadget自带的generate\_r1cs\_constraints()方法。同时添加一个约束，要求less\*1=1，也就是less必须为true。相关示例代码如下。

comparison\_gadget<FieldT> cmp(pb,9,A,B,less,less\_or\_eq,"cmp");

cmp.generate\_r1cs\_constraints();

pb.add\_r1cs\_constraint(r1cs\_constraint<FieldT>(less,1,FiedlT::one()));

最后输入witness（秘密值A），比如令A=18，这里还需要调用该Gadget的generate\_r1cs\_witness方法。这样就完成了在不泄露秘密数字A的前提下，证明数字A小于88。

4. 基于 6.3 应用案例∶以零知识证明方式提供财富达标证明，完成具体开发

1. 构建共用文件common.hpp

由于设计两个数的比较，可以使用comparison\_gadget，而在初始设置、证明以及验证三个阶段都需要构造原型版，直接使用一个共用文件会更方便。

首先需要初始化曲线参数，并定义面包板以及外部变量和中间变量：

    //初始化曲线参数

    default\_r1cs\_gg\_ppzksnark\_pp::init\_public\_params();

    //定义面包板

    protoboard<FieldT>pb;

    //定义所有需要外部输入的变量以及中间变量

    pb\_variable<FieldT>min; //VIP需要达到的财富指标

    pb\_variable<FieldT> x; // 银行知道但不能对外公布的用户财富值

    pb\_variable<FieldT> less,less\_or\_eq; // comparison\_gadget需要用到的变量

然后使用allocate()函数，将下面各个变量与protoboard连接，相当于把各个元件插到面包板上：

    //下面将各个变量与 protoboard 连接，相当于把各个元器件插到“面包板”上。

    //allocate() 函数的第二个 string 类型变量仅是用来方便 DEBUG 时的注释，方便 DEBUG 时查看日志。

    //set\_input\_sizes(n) 用来声明与 protoboard 连接的 public变量的个数 n。

    //在这里 n = 1，表明与 pb 连接的前 n = 1 个变量是 public 的，其余都是 private 的。

    //因此，要将public的变量先与pb连接

    min.allocate(pb,"min");

    x.allocate(pb,"x");

    less.allocate(pb,"less");

    less\_or\_eq.allocate(pb,"less\_or\_eq");

之后为各个变量赋值。

构造gadget，并添加相关约束：

    //构造gadget

    comparison\_gadget<FieldT> cmp(pb,n,min,x,less,less\_or\_eq,"cmp");

    cmp.generate\_r1cs\_constraints();

    //添加一个约束，要求less\*1=1，也就是less必须为true。如果是判断小于等于，则添加less\_or\_eq\*1=1的约束

    pb.add\_r1cs\_constraint(r1cs\_constraint<FieldT>(less,1,FieldT::one()));

1. 生成证明密钥和验证密钥

在这个阶段，需要把生成的证明密钥和验证密钥输入到对应文件并存储。证明密钥是银行使用的，验证密钥是银行合作商户使用的，创建setup.cpp，如下：

首先创建面包板（这里直接调用公用函数build\_protoboard()），并生成约束。指导生成密钥对，并保存到相应的文件中：

    //生成密钥对

    const r1cs\_gg\_ppzksnark\_keypair<default\_r1cs\_gg\_ppzksnark\_pp>keypair=r1cs\_gg\_ppzksnark\_generator<default\_r1cs\_gg\_ppzksnark\_pp>(constraint\_system);

    //保存证明密钥到文件bank\_pk.raw

    fstream pk("bank\_pk.raw",ios\_base::out);

    pk<<keypair.pk;

    pk.close();

    //保存验证密钥到文件client\_vk.raw

    fstream vk("client\_vk.raw",ios\_base::out);

    vk<<keypair.vk;

    vk.close();

1. 证明方使用证明密钥和其可行解构造证明

银行收到包含证明密钥的bank\_pk.raw之后可为其用户生成零知识证明，并将生成的证明发送给合作商户。此部分代码在bank-prove.cpp中：

首先也是构造面包板并生成约束（如setup.cpp所示），然后加载证明密钥。但是与之前不同的是，这里需要先输入一个隐私数据secret。

之后生成证明，并将证明保存，如下所示：

    //生成证明

    const r1cs\_gg\_ppzksnark\_proof<default\_r1cs\_gg\_ppzksnark\_pp>

        proof=r1cs\_gg\_ppzksnark\_prover<default\_r1cs\_gg\_ppzksnark\_pp>

        (bank\_pk,pb.primary\_input(),pb.auxiliary\_input());

        //primary\_input()为公开输入，auxiliary\_input()为隐私输入

    //将生成的证明保存到bank\_proof.raw文件

    fstream pr("bank\_proof.raw",ios\_base::out);

    pr<<proof;

    pr.close();

1. 验证方使用验证密钥验证证明方发过来的密钥

合作商户收到bank\_proof.raw之后，即可进行验证。代码为client-verify.cpp：

思路与构造大致相同，但是这次是解密。首先构造面包板，加载验证密钥client\_vk.raw，再加载银行生成的证明bank\_proof.raw，最后进行验证。

    //进行验证

    bool verified=r1cs\_gg\_ppzksnark\_verifier\_strong\_IC<default\_r1cs\_gg\_ppzksnark\_pp>(client\_vk,pb.primary\_input(),bank\_proof);

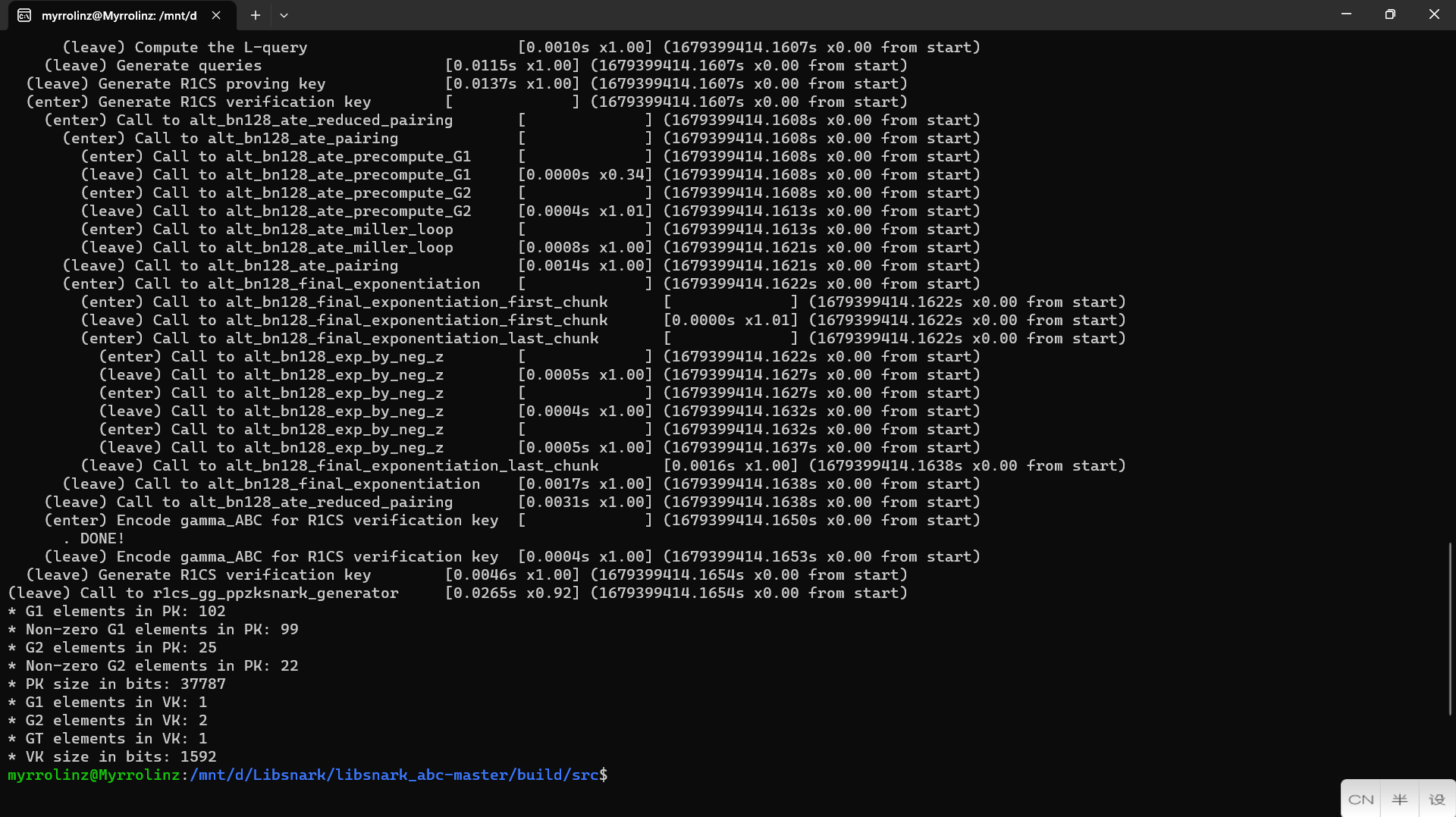
如果验证结果为1，则说明通过验证。

1. 编译运行

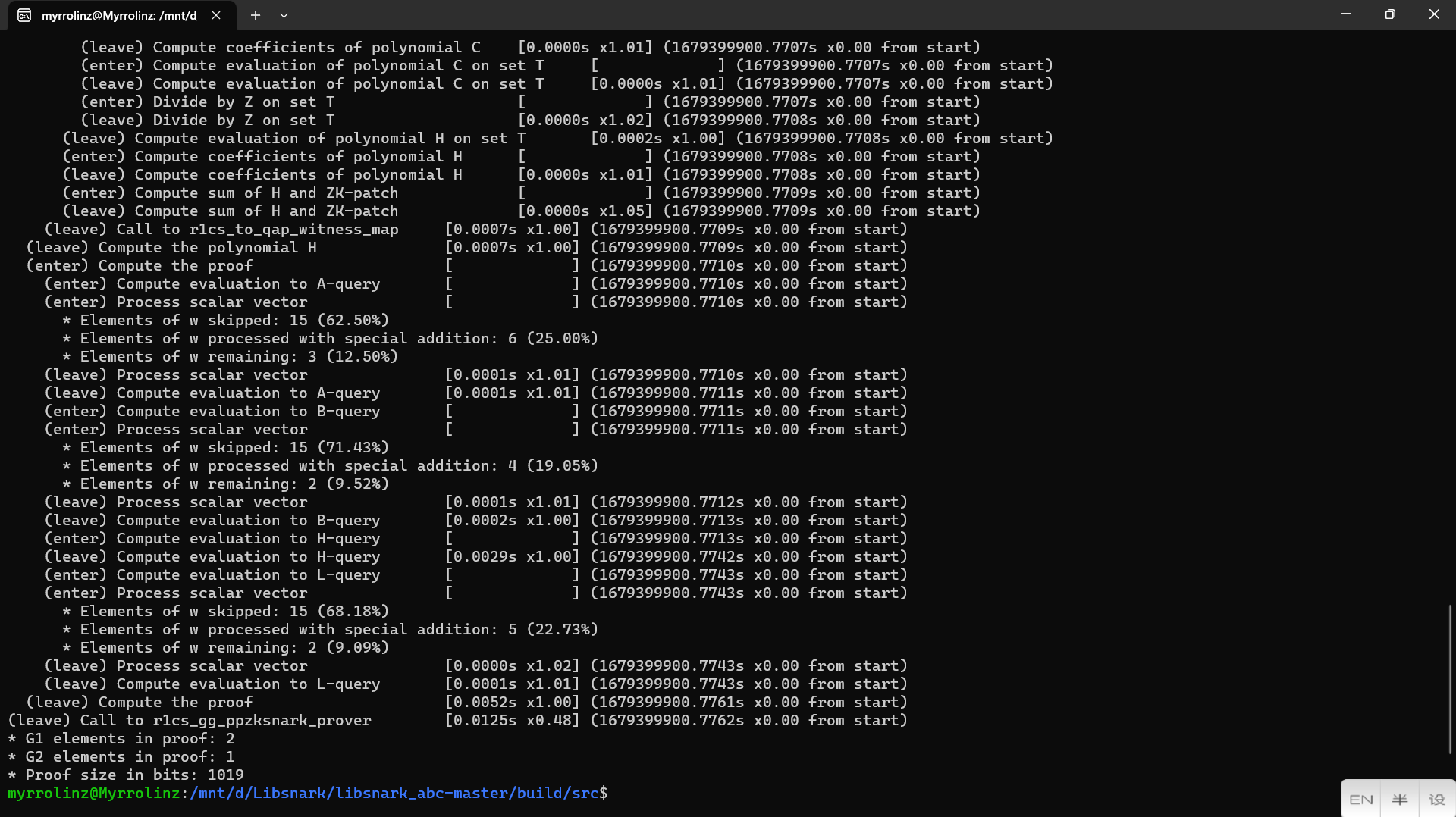
修改CMakeLists.txt文件，并分别运行三个文件。

踩坑：我的CMakeLists.txt和老师给的不一样，执行后提示缺少文件，最后还是得自己手动复制内容添加到原来的CMakeLists.txt中，之后便可运行成功。

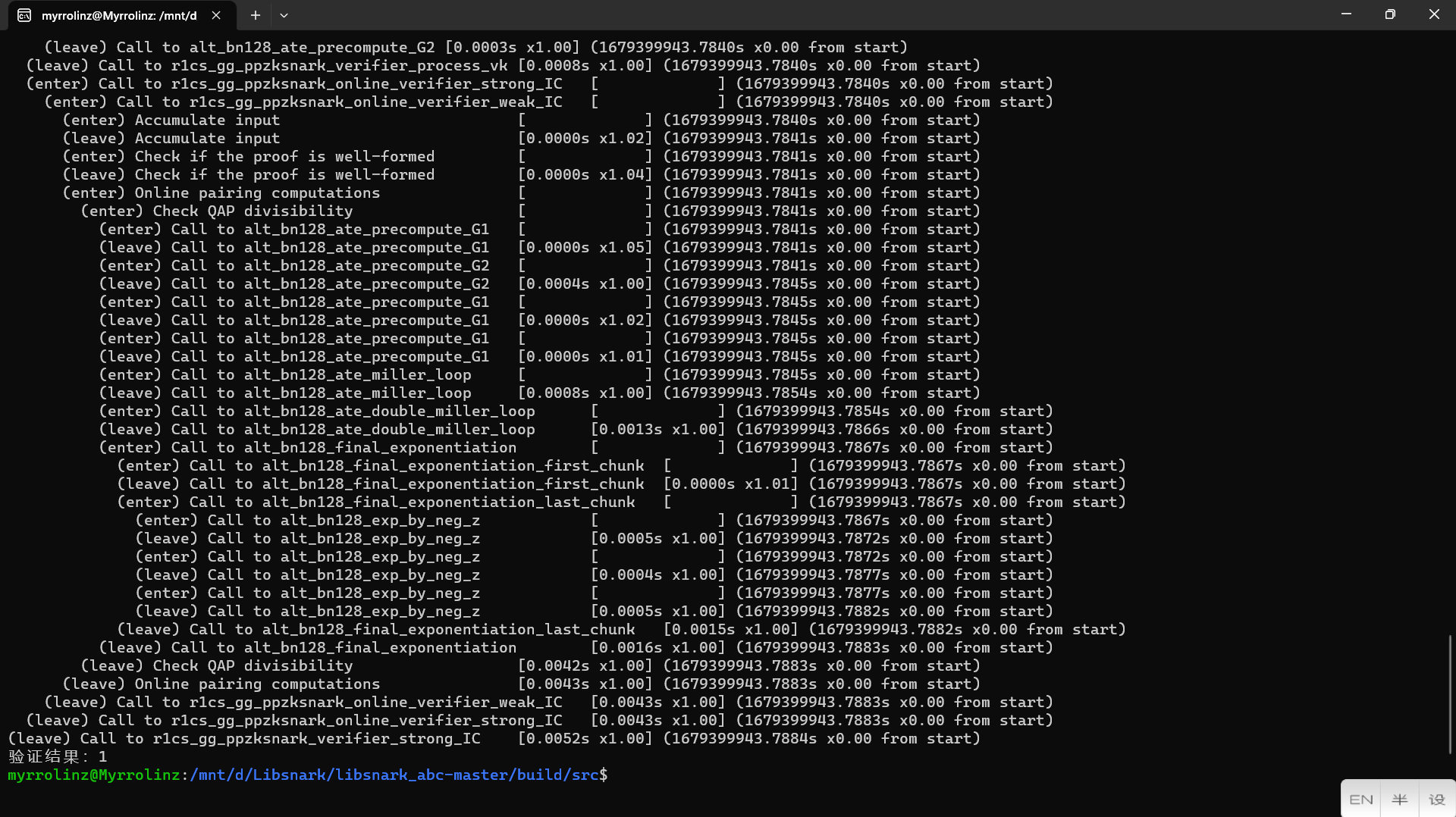
运行setup:



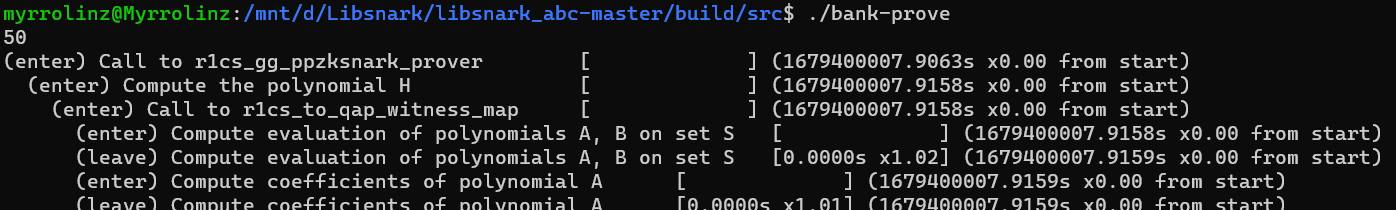
运行bank-prove，并输入100：



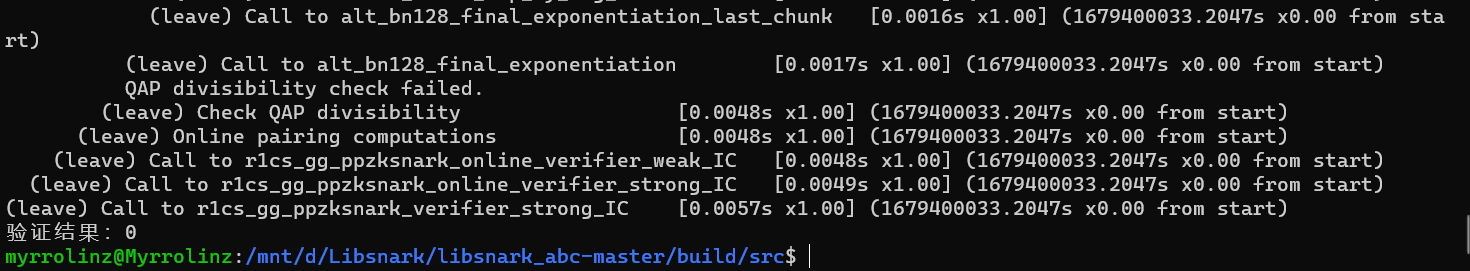
运行client-verify，结果为1，表明通过验证：



如果在运行bank-prove时输入的数为50：:



可以发现验证结果为0，则标明没能通过验证。



5. 对于方程，证明方拥有方程的解

5.1 将待证明的命题表带为R1CS

首先用算术电路表达待证明问题，对于上述问题，约束如下：

其次将算术电路拍平成多个 x=y 或 x=y(op)z 形式的等式，其中op可以是加减乘除中的一种。对于上述约束，可以拍平成如下几个等式：

最后使用原型板搭建电路。将待证明的命题用电路表示，并用 R1CS 描述电路之后，就可以构建一个protoboard。使用公用文件common.hpp，其中公有变量应设置为35。

protoboard<FieldT>build\_protoboard(int\* secret){

    //初始化曲线参数

    default\_r1cs\_gg\_ppzksnark\_pp::init\_public\_params();

    //定义面包板

    protoboard<FieldT>pb;

    //定义所有需要外部输入的变量以及中间变量

    pb\_variable<FieldT> x;

    pb\_variable<FieldT> w\_1;

    pb\_variable<FieldT> w\_2;

    pb\_variable<FieldT> w\_3;

    pb\_variable<FieldT> out;

    //下面将各个变量与 protoboard 连接，相当于把各个元器件插到“面包板”上。allocate() 函数的第二个 string 类型变量仅是用来方便 DEBUG 时的注释，方便 DEBUG 时查看日志。set\_input\_sizes(n) 用来声明与 protoboard 连接的 public变量的个数 n。在这里 n = 1，表明与 pb 连接的前 n = 1 个变量是 public 的，其余都是 private 的。因此，要将public的变量先与pb连接

    out.allocate(pb, "out");

    x.allocate(pb, "x");

    w\_1.allocate(pb, "w\_1");

    w\_2.allocate(pb, "w\_2");

    w\_3.allocate(pb, "w\_3");

pb.set\_input\_sizes(1); // 指标为可公开的值

pb.val(out)=35; // 设置输出值为35

    //为公有变量赋值

    // x\*x= w\_1

    pb.add\_r1cs\_constraint(r1cs\_constraint<FieldT>(x, x, w\_1));

    // x\*w\_1= w\_2

    pb.add\_r1cs\_constraint(r1cs\_constraint<FieldT>(x, w\_1, w\_2));

    // x+w\_2= w\_3

    pb.add\_r1cs\_constraint(r1cs\_constraint<FieldT>(x+w\_2, 1, w\_3));

    // w\_3+5=w\_4

    pb.add\_r1cs\_constraint(r1cs\_constraint<FieldT>(w\_3+5, 1, out));

    if (secret!=NULL)

    {

    pb.val(x)=secret[0];

    pb.val(w\_1)=secret[1];

    pb.val(w\_2)=secret[2];

    pb.val(w\_3)=secret[3];

    }

    return pb;

}

5.2 生成证明密钥和验证密钥

针对命题的电路已构建完毕，下面是生成公钥的初始设置阶段（Trusted Setup）。在这个阶段，需要使用生成算法为该命题生成公共参数（证明密钥和验证密钥），并把生成的证明密钥和验证密钥输出到对应文件中保存。

证明密钥供证明者使用，验证密钥供验证者使用。

Mysetup.cpp如下：

#include <libsnark/common/default\_types/r1cs\_gg\_ppzksnark\_pp.hpp>

#include <libsnark/zk\_proof\_systems/ppzksnark/r1cs\_gg\_ppzksnark/r1cs\_gg\_ppzksnark.hpp>

#include <fstream>

#include "common.hpp"

using namespace libsnark;

using namespace std;

int main()

{

    //构造面包板

    protoboard<FieldT> pb=build\_protoboard(NULL);

    const r1cs\_constraint\_system<FieldT> constraint\_system =

    pb.get\_constraint\_system();

    //生成证明密钥和验证密钥

    const r1cs\_gg\_ppzksnark\_keypair<default\_r1cs\_gg\_ppzksnark\_pp> keypair =

    r1cs\_gg\_ppzksnark\_generator<default\_r1cs\_gg\_ppzksnark\_pp>(constraint\_system);

    //保存证明密钥到文件 pk.raw

    fstream pk("pk.raw",ios\_base::out);

    pk<<keypair.pk;

    pk.close();

    //保存验证密钥到文件 vk.raw

    fstream vk("vk.raw",ios\_base::out);

    vk<<keypair.vk;

    vk.close();

    return 0;

}

5.3 证明方使用证明密钥和其可行解构造证明

在定义面包板时，我们已为 public input 提供具体数值，在构造证明阶段，证明者只需为 private input提供具体数值。再把 public input 以及 private input 的数值传给 prover 函数生成证明。生成的证明保存到proof.raw 文件中供验证者使用。

Myprove.cpp如下所示：

#include <libsnark/common/default\_types/r1cs\_gg\_ppzksnark\_pp.hpp>

#include <libsnark/zk\_proof\_systems/ppzksnark/r1cs\_gg\_ppzksnark/r1cs\_gg\_ppzksnark.hpp>

#include <fstream>

#include "common.hpp"

using namespace libsnark;

using namespace std;

int main()

{

    //输入秘密值 x

    int x;

    cin>>x;

    //为私密输入提供具体数值

    int secret[4];

    secret[0]=x;

    secret[1]=x\*x;

    secret[2]=x\*x\*x;

    secret[3]=x\*x\*x+x;

    //构造面包板

    protoboard<FieldT> pb=build\_protoboard(secret);

    const r1cs\_constraint\_system<FieldT> constraint\_system =

    pb.get\_constraint\_system();

    cout<<"公有输入："<<pb.primary\_input()<<endl;

    cout<<"私密输入："<<pb.auxiliary\_input()<<endl;

    //加载证明密钥

    fstream f\_pk("pk.raw",ios\_base::in);

    r1cs\_gg\_ppzksnark\_proving\_key<libff::default\_ec\_pp>pk;

    f\_pk>>pk;

    f\_pk.close();

    //生成证明

    const r1cs\_gg\_ppzksnark\_proof<default\_r1cs\_gg\_ppzksnark\_pp> proof =

    r1cs\_gg\_ppzksnark\_prover<default\_r1cs\_gg\_ppzksnark\_pp>(pk, pb.primary\_input(),

    pb.auxiliary\_input());

    //将生成的证明保存到 proof.raw 文件

    fstream pr("proof.raw",ios\_base::out);

    pr<<proof;

    pr.close();

    return 0;

}

5.4 验证方使用验证密钥验证证明方发来的证明

最后使用 verifier 函数校验证明。如果 verified = 1 则说明证明验证成功。

Myverify.cpp如下：

#include <libsnark/common/default\_types/r1cs\_gg\_ppzksnark\_pp.hpp>

#include <libsnark/zk\_proof\_systems/ppzksnark/r1cs\_gg\_ppzksnark/r1cs\_gg\_ppzksnark.hpp>

#include <fstream>

#include "common.hpp"

using namespace libsnark;

using namespace std;

int main()

{

    //构造面包板

    protoboard<FieldT> pb=build\_protoboard(NULL);

    const r1cs\_constraint\_system<FieldT> constraint\_system =

    pb.get\_constraint\_system();

    //加载验证密钥

    fstream f\_vk("vk.raw",ios\_base::in);

    r1cs\_gg\_ppzksnark\_verification\_key<libff::default\_ec\_pp>vk;

    f\_vk>>vk;

    f\_vk.close();

    //加载银行生成的证明

    fstream f\_proof("proof.raw",ios\_base::in);

    r1cs\_gg\_ppzksnark\_proof<libff::default\_ec\_pp>proof;

    f\_proof>>proof;

    f\_proof.close();

    //进行验证

    bool verified =

        r1cs\_gg\_ppzksnark\_verifier\_strong\_IC<default\_r1cs\_gg\_ppzksnark\_pp>

        (vk,pb.primary\_input(), proof);

    cout<<"验证结果:"<<verified<<endl;

    return 0;

}

5.5 编译运行

打开CMakeLists.txt，将以下代码添加到末尾：

add\_executable(

  mysetup

  mysetup.cpp

)

target\_link\_libraries(

  mysetup

  snark

)

target\_include\_directories(

  mysetup

  PUBLIC

  ${DEPENDS\_DIR}/libsnark

  ${DEPENDS\_DIR}/libsnark/depends/libfqfft

)

add\_executable(

  myprove

  myprove.cpp

)

target\_link\_libraries(

  myprove

  snark

)

target\_include\_directories(

  myprove

  PUBLIC

  ${DEPENDS\_DIR}/libsnark

  ${DEPENDS\_DIR}/libsnark/depends/libfqfft

)

add\_executable(

  myverify

  myverify.cpp

)

target\_link\_libraries(

  myverify

  snark

)

target\_include\_directories(

  myverify

  PUBLIC

  ${DEPENDS\_DIR}/libsnark

  ${DEPENDS\_DIR}/libsnark/depends/libfqfft

)

在~/Libsnark/libsnark\_abc-master/build下打开终端，执行以下命令：

cmake ..

make

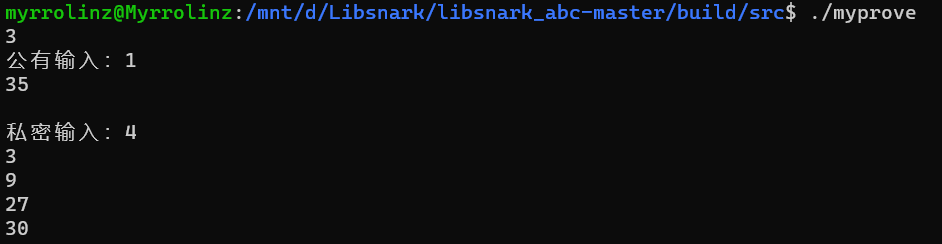
cd src

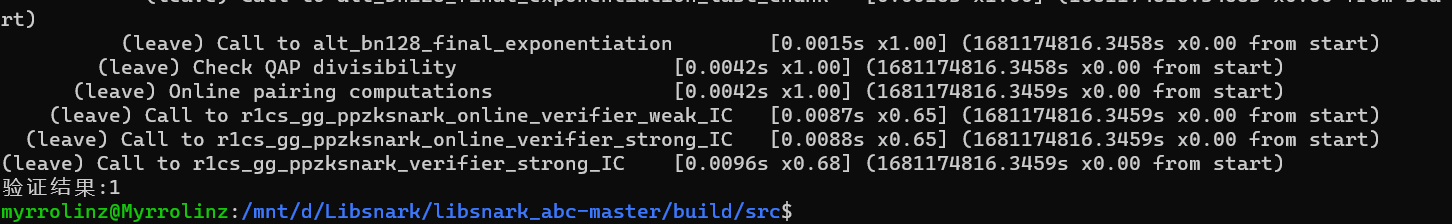
./mysetup

./myprove

3

./myverify





结果为1，验证成功。

**心得体会：**

通过本次实验，我对零知识证明有了更深的理解。由于第一次接触libsnark，而且我对密码体系了解不深，实验过程中踩过许多坑，很感谢在本次实验过程中帮助过我的同学和室友。希望通过这次实验加深对密码的了解，下一次实验能够更加顺利地进行。