南開大學
Nankai University

南　开　大　学

网络空间安全学院

数据安全实验报告

零知识证明实践

2111408 周钰宸

年级：2021 级

专业：信息安全

指导教师：刘哲理

2024 年 4 月 6 日

摘要

在日常生活中，承诺无处不在，而在安全领域承诺同样扮演着重要的角色。零知识证明就是承诺在密码学中的一个应用。是一种涉及两方或更多方的协议，允许证明者能够在不向验证者提供任何有用的信息的情况下，使验证者相信某个论断是正确的。

本次实验基于参考教材实验 3.1，利用了 1 实现简洁零知识证明 zkSNARK 的 libsnark 库，通过模拟算术电路构造 R1CS，完成了 $\mathbf{x^3 + x + 5 = out}$ 的证明生成与证明的验证。

关键字：数据安全　零知识证明　R1CS　zkSNARK　libsnark

目录

一、 实验要求

1. 实验 3.1：假设证明方有一个整数 x，他希望向验证方证明这个整数 x 的取值范围为 [0,3]。
2. 参考教材实验 3.1，假设 Alice 希望证明自己知道如下方程的解 $\mathbf{x^3 + x + 5 = out}$，其中 out 是大家都知道的一个数，这里假设 $out = 35$ 而 $x = 3$ 就是方程的解，请实现代码完成证明生成和证明的验证。

本次实验参考上述实验要求完成了**对零知识证明的应用实践**。

二、 实验原理

以下实验原理参考教材与网上资料。

(一) 零知识证明

在日常生活中，承诺无处不在。一般而言，密码学中的承诺涉及承诺方和验证方，它允许承诺方向验证方对一个秘密值做出承诺，承诺方后续会向验证方披露此秘密值。**而零知识证明就是承诺在密码学中一个十分重要的应用**。

零知识（Zero-Knowledge，ZK）是一种涉及两方或更多方的协议，允许证明者能够在不向验证者提供任何有用的信息的情况下，使验证者相信某个论断是正确的。其允许证明方让验证方相信证明方自己知道一个满足 $C(x) = 1$ 的 x，但不会进一步泄漏关于 x 的任何信息。

零知识证明具有**正确性，完备性和零知识性三个重要的性质**。在实际应用中，零知识证明通常需要将证明过程转化为验证一个谓词函数是否成立的形式。

(二) zkSNARK

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。其通常具有简洁性，无交互性，可靠性和零知识等技术特征**。

CRS 模型是在证明者构造证明之前由一个受信任的第三方产生的随机字符串，CRS 必须由一个受信任的第三方来完成，同时共享给证明者和验证者。

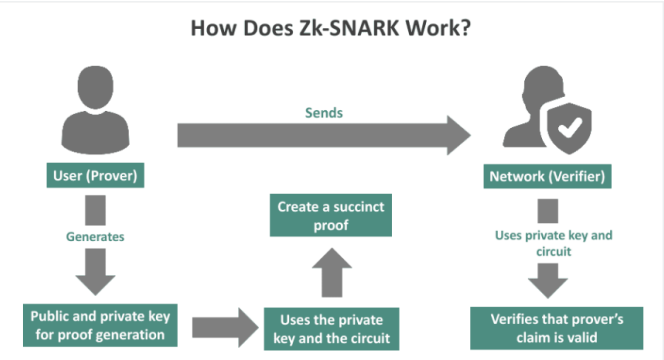

图 1: zkSNARK

zk-SNARKs 可应用于隐私保护、区块链扩容、可验证计算等领域。

1. R1CS

R1CS（Rank-1 Constraint Systems）即一阶约束系统是 zkSNARK 中的一个关键概念，**它在将数学运算电路转换为多项式时起到重要作用**。

1. **从电路到 R1CS：**
   - 在 zkSNARK 中，我们首先将真正要解决的问题（即数学运算电路）转化为多项式。
   - 这个转换过程涉及将验证者和证明者的角色分开，从而隐藏输入（原始数据）。
   - 我们需要将数学运算电路转换为一阶约束系统（R1CS）。这个过程确保验证者可以相信“输出”是“输入”通过“用作证明的数学运算电路”计算出来的。
2. **R1CS 的构建：**
   - R1CS 是由三个向量组成的序列：a、b 和 c。
   - 解向量 s 满足 $\mathrm{s.a}^{\star}\mathrm{s.b}-\mathrm{s.c}=0$，这被称为一个约束。
   - 通过拉格朗日插值法，我们将这些向量转换为多项式，从而将验证者和证明者之间的绑定实现在输出为真的前提下。
3. **从 R1CS 到多项式：**
   - 我们将 R1CS 中的向量 a 转换为多项式。
   - 通过这个转换，我们将证明多个原方程转换为证明多个 $\mathrm{s.a}^{\star}\mathrm{s.b}-\mathrm{s.c}=0$。
   - 这些多项式与 R1CS 是等价的。

   因此 **R1CS 帮助我们将数学运算电路转换为多项式，从而实现零知识证明的目标。后面的实验中将会重点体现这一关键步骤**。

2. libsnark

libsnark 是一个用于开发 zk-SNARKs 应用的 C++ 代码库。它由 SCIPR Lab 开发并维护，是实现 zk-SNARKs 电路的关键框架之一。

1. libsnark 实现了 zkSNARK 方案，将计算的完整性以零知识的方式证明和验证。
2. 它支持通用的证明系统，包括**预处理 zkSNARK（preprocessing zkSNARK）。即 ppzksnark**。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**。
3. **Groth16 是最广泛使用的 libsnark 证明系统之一**。其计算分为三个部分：
   - 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。

本次实验，也会重点使用 **Groth16 作为证明系统实现零知识证明**。

三、 实验准备

(一) 实验环境

首先本次实验由于需要用到 Linux 操作系统，这里说明一下我所使用的虚拟机和对应的操作环境：

虚拟机软件	VMware Workstation 17 Pro
宿主机	Windows 11 家庭中文版
虚拟机操作系统	Ubuntu 20.04 LTS

表 1: 本次实验环境及工具

(二) libsnark 环境搭建

1. 下载必备资源

Libsnark 的环境配置需要对其多个子模块进行编译安装。第一部分首先需要下载所有压缩包，具体流程如下：

- 创建名为 Libsnark 的文件夹
- 打开 https://github.com/secbit/libsnark_abc，点击“Code”、“Download ZIP”，下载后解压到 Libsnark 文件夹，得到 /Libsnark/libsnark_abcmaster
- 打开 https://github.com/sciprlab/libsnark，点击“Code”、“Download ZIP”，下载解压后，将其中文件复制到 / Libsnark/libsnark_abcmaster/depends/libsnark 文件夹内
- 打开 https://github.com/sciprlab/libsnark，点击“depends”，可以看到六个子模块的链接地址，分别下载 ZIP。
- 分别点击这六个链接并下载解压，得到如下六个文件夹。**这里将六个文件夹分别命名为 Libfqfft、Libff、Gtest、Xbyak、Atepairing、Libsnarksupercop**。然后根据对应的 Linux 操作系统，由于我是 Ubuntu 20.04 LTS，因此通过以下命令进行第一步的安装：

```
sudo apt install build-essential cmake git libgmp3-dev libprocps-dev
python3-markdown libboost-program-options-dev libssl-dev python3 pkgconfig
```

完成后得到如下图所示结果：

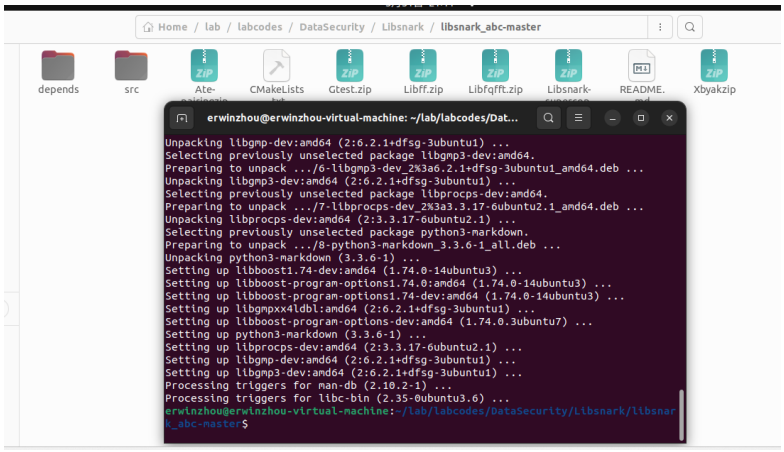

图 2: libsnark 模块安装结果

如图2所示完成了第一步对下载资源的解压和安装了。**接下来要分别对六个子模块分别进行安装**。

2. 子模块安装

1. **Xbyak：**将下载得到的文件夹 Xbyak 内的文件复制到

   /Libsnark/libsnark_abcmaster/depends/libsnark/depends/xbyak,并在该目录下打开终端，执行以下命令

```
...$ sudo make install
mkdir -p /usr/local/include/xbyak
cp -pR xbyak/*.h /usr/local/include/xbyak
```

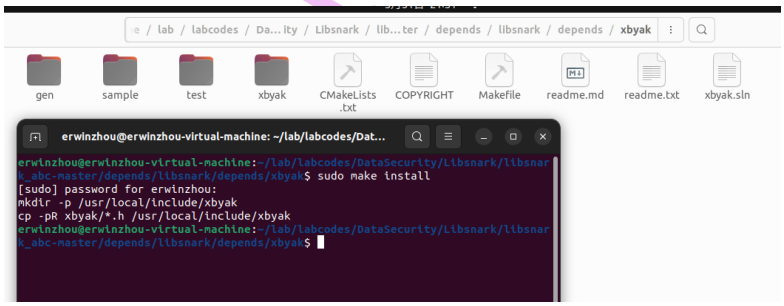

图 3: xbyak 安装结果

如图3所示，**出现了和参考书上一致的代码，证明安装成功**。

2. **Ate-pairing** 将下载得到的文件夹 Ate-pairing 内的文件复制到

   /Libsnark/libsnark_abc-master/depends/libsnark/depends/ate-pairing，并在该目录下打开终端，执行以下命令:

```
...$ make -j
.../test$ ./bn
...
```

```
err=0(test=461)
```

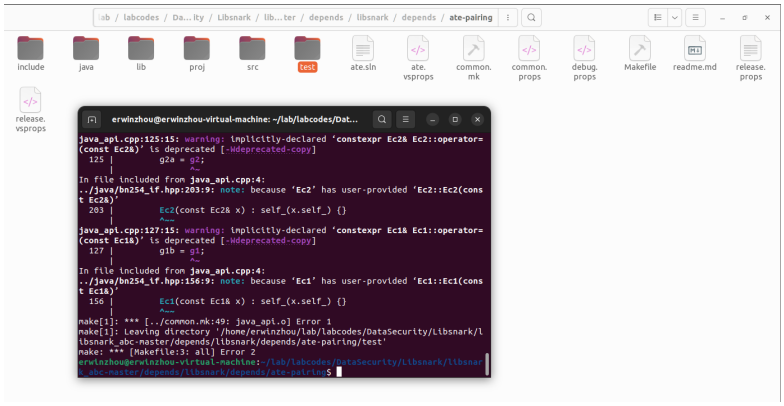

图 4: ate-pairing 安装结果 1

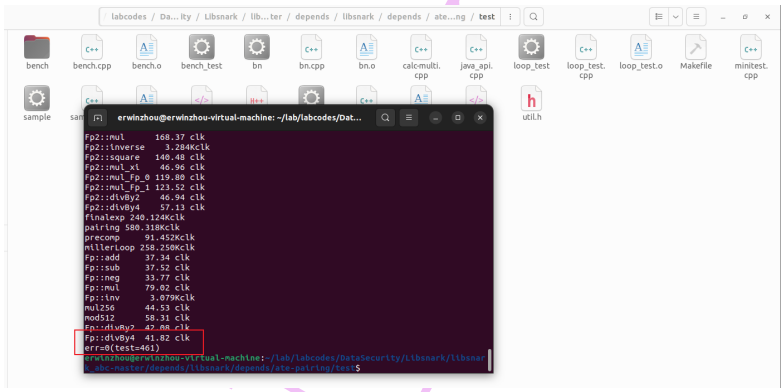

图 5: ate-pairing 安装结果 2

如4和5所示，**出现了和参考书上一致的代码，证明安装成功**。

3. **Libsnark-supercop：**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，并在该目录下打开终端，执行以下命令：

```
...$ ./do
ar: creating ../lib/libsupercop.a
```

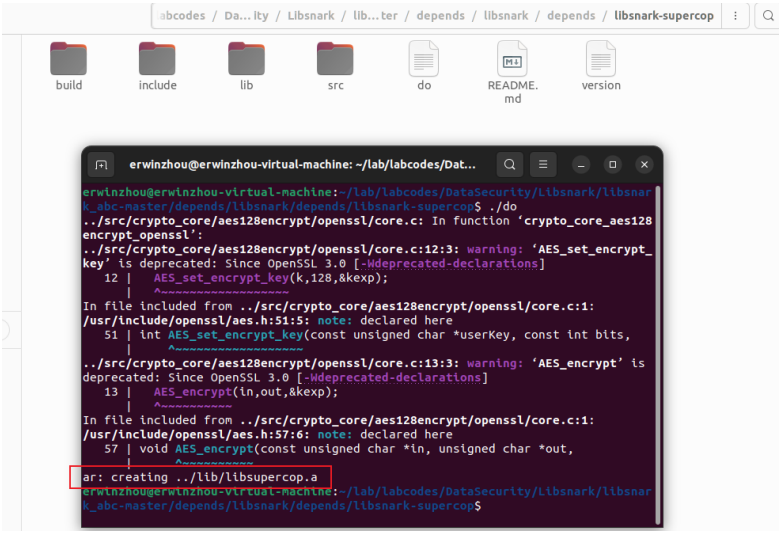

图 6: Libsnark-supercop 安装结果

如6所示，**出现了和参考书上一致的代码，证明安装成功**。

4. **GTest：**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

5. **Libff：**将下载得到的文件夹 Libff 内的文件复制到

   /Libsnark/libsnark_abc-master/depends/libsnark/depends/libff。

   点击 libff->depends，可以看到一个 ate-pairing 文件夹和一个 xbyak 文件夹，这是 libff 需要的依赖项。打开这两个文件夹，会发现它们是空的，这时候需要将下载得到的 Ate-pairing 和 Xbyak 内的文件复制到这两个文件夹下。

   之后在 /Libsnark/libsnark_abc-master/depends/libsnark/depends/libff 下打开终端，执行命令:

```
...$mkdir build
...$cd build
...$cmake ..
...$make
...$sudo make install
```

安装完之后检测是否安装成功，执行以下命令:

```
...$make check
100% tests passed, 0 tests failed out of 3
Total Test time (real) = 0.04 sec
[100%] Built target check
```

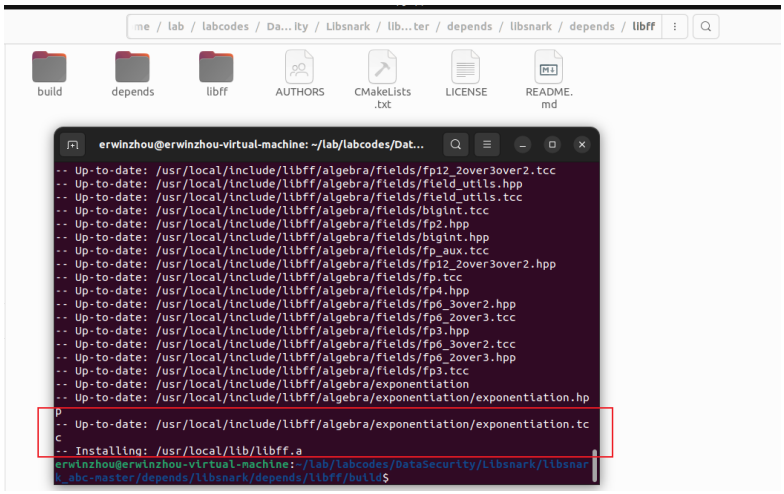

图 7: libff 安装结果

如图7所示，**出现了和参考书上一致的代码，证明安装成功**。

6. **Libfqfft：**将下载得到的文件夹 Libfqfft 内的文件复制到

   /Libsnark/libsnark_abc-master/depends/libsnark/depends/libfqfft。

   点击 libfqfft->depends，可以看到 libfqfft 有四个依赖项，分别是 ate-pairing、gtest、libff、xbyak，点开来依然是空的。和上一步一样，将下载得到的文件夹内文件复制到对应文件夹下。注意 libff 里还有 depends 文件夹，里面的 ate-pairing 和 xbyaky 也是空的，需要将下载得到的 airing 和 Xbyak 文件夹内的文件复制进去。

   在 /Libsnark/libsnark_abc-master/depends/libsnark/depends/libfqfft 下打开终端，执行命令：

```
...$mkdir build
...$cd build
...$cmake ..
...$make
...$sudo make install
```

   安装完之后检测是否安装成功，执行以下命令：

```
...$make check
100% tests passed, 0 tests failed out of 3

Total Test time (real) = 0.04 sec
[100%] Built target check
```

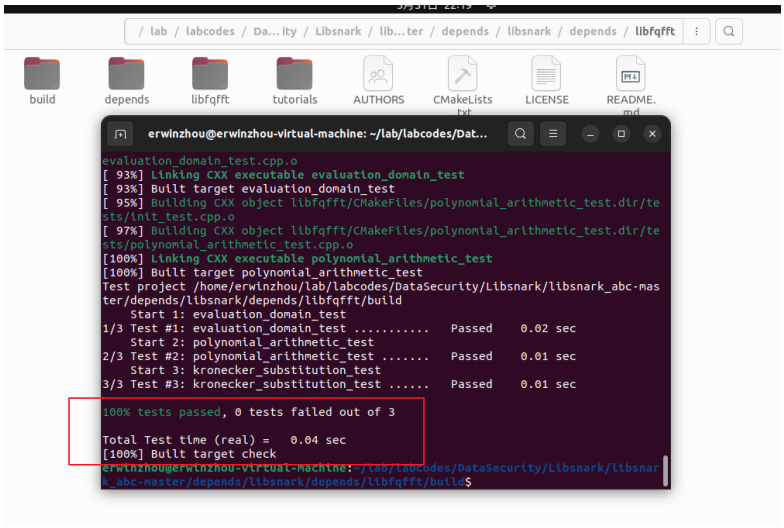

图 8: libfqfft 安装结果

如图8所示，**出现了和参考书上一致的代码，证明安装成功。**

接下来就是进行 **Libshark 的编译安装和整体的编译安装。**在 /Libsnark/libsnar_abcmaster/depends/libsnark 下打开终端，执行以下命令:

```
...$mkdir build
...$cd build
...$cmake ..
...$make
...$make check
```

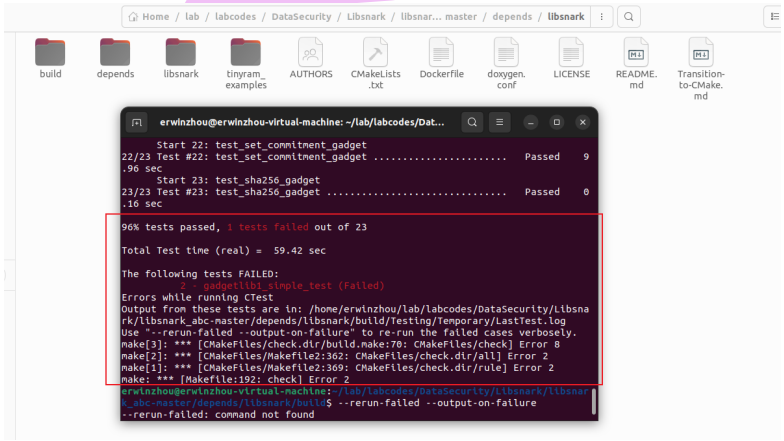

图 9: libshark 安装结果

如图9所示，**这里实际上出现了和参考书不一致的结果，出现了一个错误。但在我重复了两遍安装过程却得到了相同的结果。在这之前我也出现了 MAKE ERROR 的情况，因此我推测可能和我的 Ubuntu 一些固有配置有关。我于是继续了实验，最后发现这样并没有影响我最终的实验验证。**

最后一步在/Libsnark/libsnar_abcmaster 打开终端，然后执行如下命令:

```
...$mkdir build
...$cd build
...$cmake ..
...$make
```

最后**运行程序，使用如下命令：**

```
...$./src/test
```

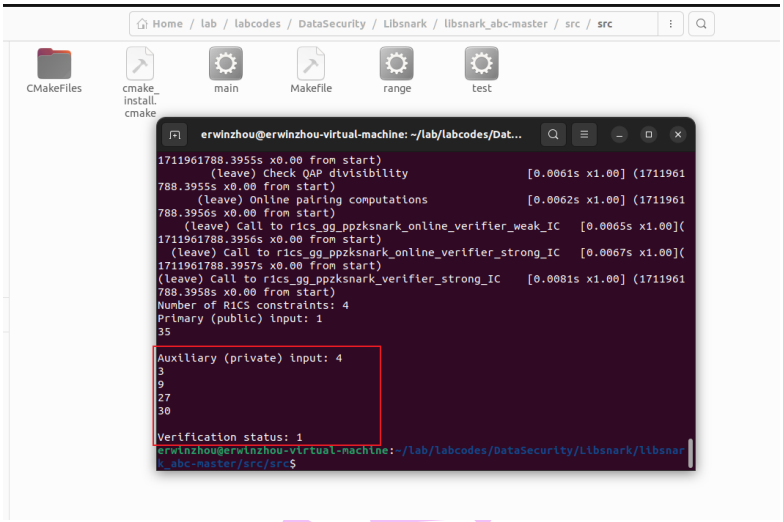

图 10: 整体编译安装结果

如图10所示，整体编译安装结果**与参考书结果一致，证明上一步的不一致没有产生太多的影响，这里就继续展开实验了。**

四、 实验过程

(一) 将待证明的命题表达为 R1CS

1. 用算术电路表示待证明问题

本次实验将会遵循一般的开发 zk-SNARKs 应用的步骤，将方程转换为算术电路的形式。简单地说，算术电路以变量或数字作为输入，并且允许使用加法、乘法两种运算来操作表达式。**因此第一步我们需要定义变量：**

```
    pb_variable<FieldT> x;
    pb_variable<FieldT> sym_1;
    pb_variable<FieldT> y;
    pb_variable<FieldT> sym_2;
    pb_variable<FieldT> out;
```

2. 用 R1CS 描述电路

将待证明命题转换为算术电路之后，就可以将证明过程转换为证明，**这需要我们将算数电路拍平成多个** $x = y$，$x = y(op)z$ **形式的等价。其中可以是加、减、乘、除运算符中的一种。**

1. **输入：变量** x。
2. **操作：**
   - $x * x = sym_1$。
   - $sym_1 * x = y$。
   - $y + x = sym_2$。
   - $sym_2 + 5 = out$。

3. 使用原型板搭建电路

然后，使用原型板 protoboard 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。**protoboard，也就是原型板或者面包板，可以用来快速搭建算术电路，把所有变量、组件和约束关联起来。**

接下来我们参考实验 3.1 的程序，将我们的程序也分为四个部分，具体而言：

- **common.hpp**
- **mysetup.hpp**
- **myprove.hpp**
- **myverify.hpp**

然后我们将会依次构造这四个程序部分，并按照和参考书中一致的流程进行编写：

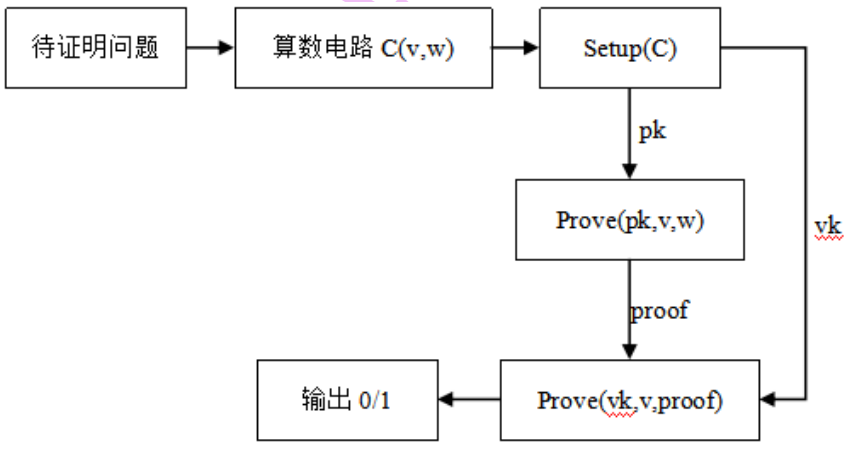

图 11: pipeline

1. **common.hpp**: 因为在初始设置、证明、验证三个阶段都需要构造面包板，**所以这里将下面的代码放在一个公用的文件 common.hpp 中供三个阶段使用。**

```
#include <libsnark/common/default_types/r1cs_gg_ppzksnark_pp.hpp>
#include <libsnark/zk_proof_systems/ppzksnark/
r1cs_gg_ppzksnark/r1cs_gg_ppzksnark.hpp>
#include <libsnark/gadgetlib1/pb_variable.hpp>
using namespace libsnark;
using namespace std;
constexpr auto primary_input = 35;
typedef libff::Fr<default_r1cs_gg_ppzksnark_pp> FieldT; // Define the
    finite field to be used
protoboard<FieldT> build_protoboard(int *secret) // Define the function
    to build the protoboard
{
  // Initialize the curve parameters
  default_r1cs_gg_ppzksnark_pp::init_public_params();
  // Create the protoboard
  protoboard<FieldT> pb;
  // Define all the variables that need external input and intermediate
      variables
  pb_variable<FieldT> x;
  pb_variable<FieldT> sym_1;
  pb_variable<FieldT> y;
  pb_variable<FieldT> sym_2;
  pb_variable<FieldT> out;
  // Connect the variables to the protoboard, similar to plugging in
      components on a breadboard. The second string variable in the
      allocate() function is only used for debugging comments.
  out.allocate(pb, "out");
  x.allocate(pb, "x");
  sym_1.allocate(pb, "sym_1");
  y.allocate(pb, "y");
  sym_2.allocate(pb, "sym_2");
  // Declare the number of public variables, set_input_sizes(n) is used
      to declare the number of public variables connected to the
      protoboard. Here, n = 1, indicating that the first n = 1 variables
      connected to pb are public, and the rest are private. Therefore, the
      public variables need to be connected to pb first (out is public).
  pb.set_input_sizes(1);
  // Assign a value to the public variable
  pb.val(out) = primary_input;
  // At this point, all variables have been successfully connected to the
      protoboard. The next step is to define the constraints between these
      variables.

```

```
33   // Add R1CS constraints to protoboard
34
35   // x*x = sym_1
36   pb.add_r1cs_constraint(r1cs_constraint<FieldT>(x, x, sym_1));
37
38   // sym_1 * x = y
39   pb.add_r1cs_constraint(r1cs_constraint<FieldT>(sym_1, x, y));
40
41   // y + x = sym_2
42   pb.add_r1cs_constraint(r1cs_constraint<FieldT>(y + x, 1, sym_2));
43
44   // sym_2 + 5 = ~out
45   pb.add_r1cs_constraint(r1cs_constraint<FieldT>(sym_2 + 5, 1, out));
46
47   // During the proof generation phase, the prover passes in the secret
         inputs and assigns values to the secret variables, otherwise they
         are set to NULL
48   if (secret != NULL)
49   {
50     pb.val(out) = secret[0];
51
52     pb.val(x) = secret[1];
53     pb.val(sym_1) = secret[2];
54     pb.val(y) = secret[3];
55     pb.val(sym_2) = secret[4];
56   }
57   return pb;
58 }
```

对代码一些关键部分解析如下:

(a) **引入库和命名空间声明**: 这部分代码引入了必要的库文件, 并使用 using namespace 声明了 libsnark 和 std 命名空间, 以便在后续代码中方便地使用这些库和标准库的功能。同时引用了三个头文件: 第一个头文件是为了引入 default_r1cs_gg_ppzksnark_pp 类型; 第二个则为了引入证明相关的各个接口; 第三个头文件是 pb_variable, 用来定义电路相关的变量。

(b) **全局变量和类型定义**: 定义了全局变量和类型, 这为构建原型板和有限字段设置了基础。

- constexpr auto primary_input = 35; 定义了一个常量, 表示公开输入的值。
- typedef libff::Fr<default_r1cs_gg_ppzksnark_pp> FieldT; 定义了使用的有限字段类型。

(c) **构建原型板函数定义**: 这是代码中最核心的部分, 定义了一个函数 build_protoboard, 它接收一个秘密输入并返回一个配置好的原型板对象。

i. **初始化曲线参数**：通过 default_r1cs_gg_ppzksnark_pp::init_public_params(); 调用来初始化公开参数。

ii. **创建原型板**：实例化一个原型板对象 pb。

iii. **声明和分配变量**：声明原型板上的变量，并将它们与原型板关联。这包括公开输入变量 out 和私有变量 x、sym_1、y、sym_2。

iv. **设置公开输入大小**：通过 pb.set_input_sizes(1); 声明原型板上的公开输入变量数量。

v. **定义约束**: 为原型板添加一系列约束, 这些约束定义了变量之间的关系, 如 $x \times x = sym_1$ 和 $sym_1 \times x = y$ 等。

vi. **处理秘密输入**：如果提供了秘密输入，将这些值赋给原型板上的对应变量。
在处理零知识证明的代码中，当我们有一个非空的 secret 参数时，这表示我们正处于证明的创建阶段。在这个阶段，我们需要给那些不公开的变量分配具体的值。这个赋值过程是通过将 secret 数组中的元素依次赋值给原型板中定义的变量来完成的，例如，secret 数组的第一个元素赋给 x，第二个元素赋给 sym_1，以此类推。
反过来，如果 secret 是空的，那么我们认为现在是在验证一个已存在的证明，这时候就不需要再给这些私密变量赋新的值，直接使用原型板进行验证即可。

2. **mysetup.hpp: 生成证明密钥和验证密钥**。

至此，针对命题的电路已构建完毕。接下来，是生成公钥的初始设置阶段（Trusted Setup）。在这个阶段，我们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。**这段代码和参考教材保持一致即可，无需改动。其中，证明密钥供证明者使用，验证密钥供验证者使用。**

```
#include "common.hpp"
#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>
using namespace libsnark;
using namespace std;
int main()
{
    // Construct the protoboard
    protoboard<FieldT> pb = build_protoboard(nullptr);
    const r1cs_constraint_system<FieldT> constraint_system =
        pb.get_constraint_system();

    // Generate the proving key and verification key
    const r1cs_gg_ppzksnark_keypair
    <default_r1cs_gg_ppzksnark_pp> keypair =
        r1cs_gg_ppzksnark_generator
        <default_r1cs_gg_ppzksnark_pp>(constraint_system);

    // Save the proving key to a file named "pk.raw"
```

```
    fstream pk("pk.raw", ios_base::out);
    pk << keypair.pk;
    pk.close();

    // Save the verification key to a file named "vk.raw"
    fstream vk("vk.raw", ios_base::out);
    vk << keypair.vk;
    vk.close();

    return 0;
}
```

3. **myprove.hpp：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。

```
#include "common.hpp"
#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 <cmath>
using namespace libsnark;
using namespace std;
int main()
{
    // Provide specific values for private inputs
    double t = (5 - primary_input) / 2.0;
    double delta = sqrt(t * t + 1.0 / 27.0);
    double res = cbrt(-t + delta) - cbrt(t + delta);
    int x = round(res);
    int secret[5];
    secret[0] = primary_input;
    secret[1] = x;
    secret[2] = x * x;
    secret[3] = x * x * x;
    secret[4] = x * x * x + x;
    // Build the protoboard
    protoboard<FieldT> pb = build_protoboard(secret);
    const r1cs_constraint_system<FieldT>
    constraint_system = pb.get_constraint_system();
    cout << "Public input: " << pb.primary_input() << endl;
    cout << "Private input: " << pb.auxiliary_input() << endl;
```

```
    // Load the proving key
    fstream f_pk("pk.raw", ios_base::in);
    r1cs_gg_ppzksnark_proving_key<libff::default_ec_pp> pk;
    f_pk >> pk;
    f_pk.close();
    // Generate the proof
    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());
    // Save the generated proof to proof.raw file
    fstream pr("proof.raw", ios_base::out);
    pr << proof;
    pr.close();
    cout << pb.primary_input() << endl;
    cout << pb.auxiliary_input() << endl;
    return 0;
}
```

对代码一些关键部分解析如下:

(a) **引入必要的库和命名空间**: 这部分包括 libsnark 的核心功能库、C++ 的文件流和数学库，以及刚刚定义的头文件 common.hpp。

(b) **私密输入的计算**: 主函数开始先计算一组基于公开输入的私密输入值。**这一部分也是我们的重点更改的地方**。

这里为了**呼应在 common.hpp，**通过 pb 的 val，将 x，sym_1，y 和 sym_2 赋值于在此处定义的 secret 数组。**定义零知识证明的过程中，公开输入和私密输入进行了区分和一一对应**。

这种一一对应关系确保了每个私密输入的计算结果都能正确地赋值给相应的原型板变量。如果这种对应关系不明确或错误，那么生成的证明将无法正确反映证明者的意图，进而无法通过验证。它不仅保证了证明的生成和验证能够基于一致的输入进行，而且还保护了私密输入的安全，确保这些信息不会在证明过程中被不必要地暴露。

(c) **构建原型板**: 通过调用 build_protoboard 函数并传入计算得到的私密输入，初始化一个原型板。这个原型板将用于定义和存储零知识证明的数学约束。

(d) **输出公开和私密输入**: 代码接着利用原型板对象的方法输出公开输入和私密输入，这有助于验证和调试。

(e) **载入证明密钥**: 接下来代码从文件中读取证明密钥（proving key），这是生成有效证明的必要条件。

(f) **生成并保存证明**: 使用原型板的公开和私密输入，结合之前载入的证明密钥，生成零知识证明。然后将这个证明写入到一个文件中，以便于后续的分发或验证。

4. **myverify.hpp：验证方使用验证密钥验证证明方发过来的证明**。

最后我们使用 verifier 函数校验证明。如果 **verified = 1 则说明证明验证成功。这部分也和参考教材代码保持一致即可，无需更改。**

```
#include "common.hpp"
#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>
using namespace libsnark;
using namespace std;
int main()
{
    // Build the protoboard
    protoboard<FieldT> pb = build_protoboard(NULL);
    const r1cs_constraint_system<FieldT> constraint_system =
        pb.get_constraint_system();
    // Load the verification key
    fstream f_vk("vk.raw", ios_base::in);
    r1cs_gg_ppzksnark_verification_key<libff::default_ec_pp> vk;
    f_vk >> vk;
    f_vk.close();
    // Load the proof generated by the bank
    fstream f_proof("proof.raw", ios_base::in);
    r1cs_gg_ppzksnark_proof<libff::default_ec_pp> proof;
    f_proof >> proof;
    f_proof.close();
    // Perform verification
    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;
}
```

在进行了上述四个模块的编写后，为了**能够一致性地进行编译和运行，我们参考教材中进行了 CMakeLists.txt 的设计**。位置位于 /Libsnark/libsnark_abc-master/src 目录下。

```
include_directories(.)

add_executable(
  main

  main.cpp
)
target_link_libraries(
```

```
  main

  snark
)
target_include_directories(
  main

  PUBLIC
  ${DEPENDS_DIR}/libsnark
  ${DEPENDS_DIR}/libsnark/depends/libfqfft
)

add_executable(
  test

  test.cpp
)
target_link_libraries(
  test

  snark
)
target_include_directories(
  test

  PUBLIC
  ${DEPENDS_DIR}/libsnark
  ${DEPENDS_DIR}/libsnark/depends/libfqfft
)

add_executable(
  range

  range.cpp
)
target_link_libraries(
  range

  snark
)
target_include_directories(
  range

  PUBLIC
```

```
  ${DEPENDS_DIR}/libsnark
  ${DEPENDS_DIR}/libsnark/depends/libfqfft
)

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
```

```
97  ${DEPENDS_DIR}/libsnark/depends/libfqfft
98  )
```

1. **设置包含目录**：通过 include_directories(.) 命令，将当前目录设置为编译器查找头文件的包含目录之一。
2. **添加可执行文件**：使用 add_executable 命令为项目中的每个应用程序定义一个可执行文件目标，包括 main, test, range, mysetup, myprove, 和 myverify，每个目标都关联了相应的.cpp 源文件。
3. **链接库文件**: 对于每个可执行文件目标, target_link_libraries 命令用于链接 snark 库，这一步骤确保了在链接阶段能解决源文件中对 snark 库符号的外部引用。
4. **设置目标包含目录**: target_include_directories 为每个目标设置了额外的包含目录, 指向 libsnark 库及其依赖项 libfqfft 的路径，这些路径通过变量 ${DEPENDS_DIR} 指定，以确保编译时能找到这些库的头文件。
5. **公共包含路径**：通过在 target_include_directories 中使用 PUBLIC 关键字，使得这些包含目录不仅对目标本身可见，也对任何链接了这些目标的其他目标可见，从而自动获得对这些额外包含目录的访问权限。

最后我们只需要依次使用**如下命令在 /Libsnark/libsnark_abc-master/build 的终端下，便可以完成程序的验证：**

```
cmake ..
make
cd src
./mysetup
./myprove
2
./myverify
```

```
Consolidate compiler generated dependencies of target snark_supercop
[ 52%] Built target snark_supercop
Consolidate compiler generated dependencies of target ff
[ 71%] Built target ff
Consolidate compiler generated dependencies of target snark
[ 88%] Built target snark
Consolidate compiler generated dependencies of target snark_adsnark
[ 90%] Built target snark_adsnark
Consolidate compiler generated dependencies of target main
[ 90%] Built target main
Consolidate compiler generated dependencies of target test
[ 92%] Built target test
Consolidate compiler generated dependencies of target range
[ 94%] Built target range
[ 96%] Building CXX object src/CMakeFiles/mysetup.dir/mysetup.cpp.o
[ 96%] Linking CXX executable mysetup
[ 96%] Built target mysetup
[ 98%] Building CXX object src/CMakeFiles/myprove.dir/myprove.cpp.o
[ 98%] Linking CXX executable myprove
[ 98%] Built target myprove
[100%] Building CXX object src/CMakeFiles/myverify.dir/myverify.cpp.o
[100%] Linking CXX executable myverify
[100%] Built target myverify
```

图 12: cmake 结果

图12展示了 cmake 的结果。

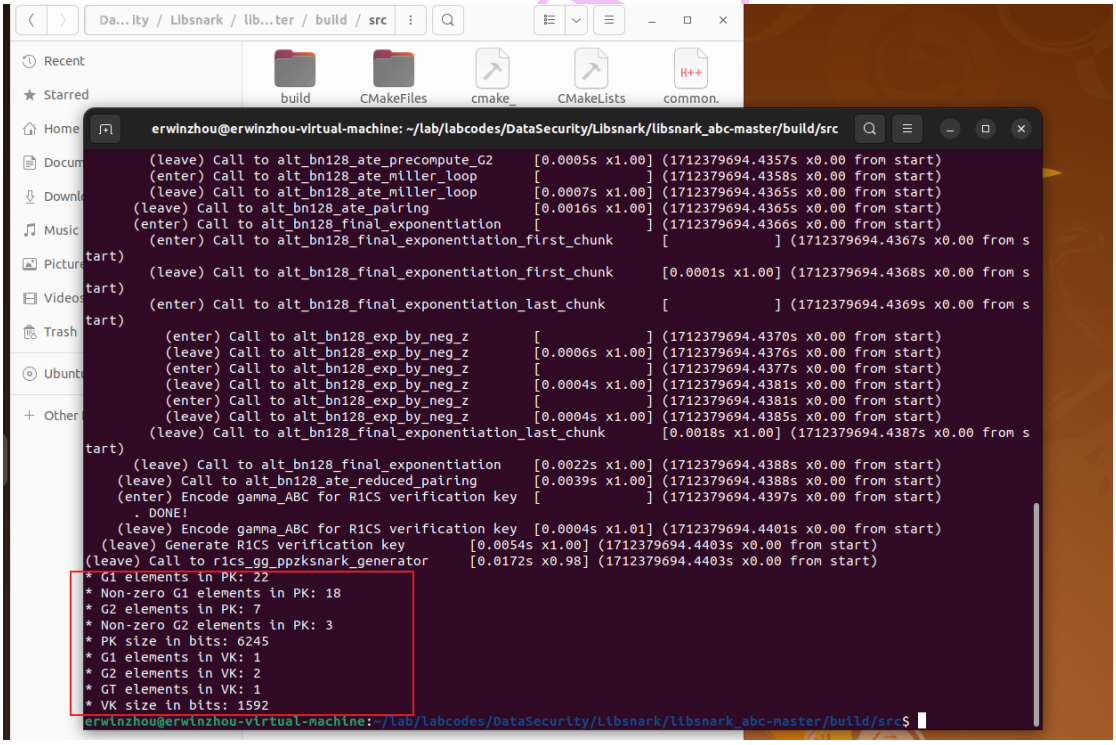

图 13: mysetup 运行结果

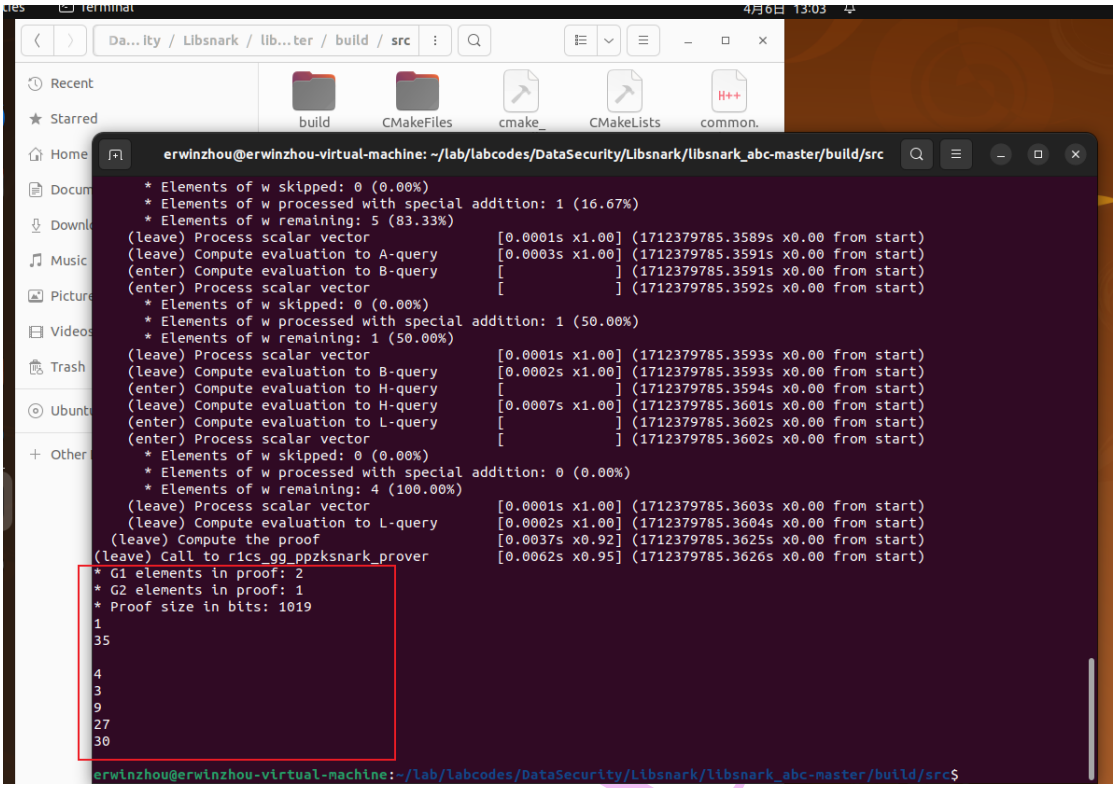

图 14: myprove 运行结果

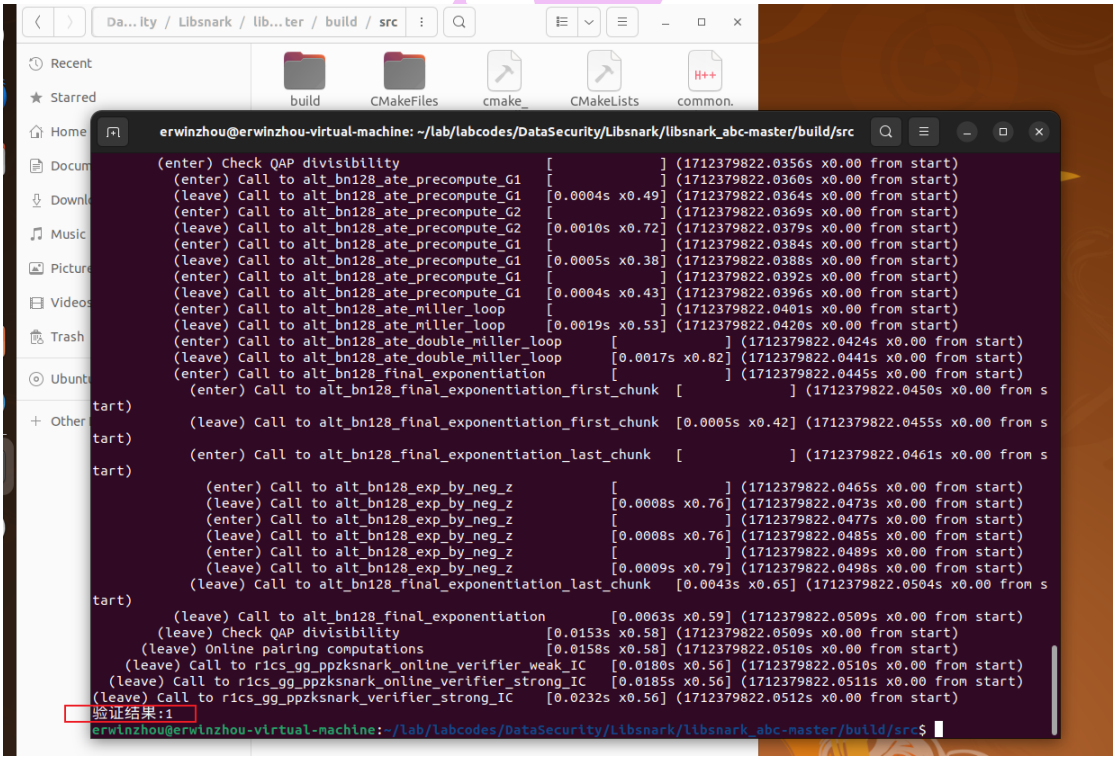

图 15: myverify 运行结果

图13，14和15依次展现了 **mysetup，myprove 和 myverify 三个可执行文件的运行结果**。

可以看到最终成功输出了”验证结果：1“的文字。因此针对于本次实验命题：假设 Alice 希望证明自己知道如下方程的解 $\mathbf{x^3 + x + 5 = out}$, 其中 out 是大家都知道的一个数, 这里假设 $out = 35$ 而 $x = 3$ 就是方程的解。**便实现了完美的验证。 到此实验全部结束，非常成功！**

五、 实验总结与思考

本次实验我通过将课堂上老师讲授的零知识证明的知识在课后进一步进行了复习。跟着参考资料和老师讲解视频的思路，对参考代码进行了研读。

由此**我基于教材中的例子，完成了本次的实验内容**。工作包括：

1. 尽管多次受阻，**但反复配置环境 libsnark，搭建了实验平台**。
2. 熟悉了 Linux 环境中**使用强大的 CMakeLists 来进行编译配置的便捷**。
3. 熟悉了 **libsnark 库**进行简洁零知识证明 zkSNARK 的方式。
4. 基于参考代码，**自己编写完成了对应的实验练习，解决了遇到的问题和困难。让我对零知识证明的原理理解更加深刻**。

总的来说，本次实验我收获颇丰，希望在后面的实验中继续努力，将数据安全领域的知识熟记于心，并在实验中反复巩固，不断探索。

参考文献