

**《数据安全》课程实验报告**

**实验四：零知识证明实践**

****

学 院 网络空间安全学院

专 业 信息安全

学 号 2112060

姓 名 孙蕗

目录

[一、 实验要求 3](#_Toc14785)

[二、 实验过程 3](#_Toc24271)

[（一） 安装libsnark框架 3](#_Toc13075)

[（二） 将待证明的命题表达为R1CS 15](#_Toc17259)

[（三） 生成证明密钥和验证密钥 20](#_Toc23808)

[（四） 证明方使用证明密钥和其可行解构造证明 21](#_Toc9963)

[（五） 验证方使用验证密钥验证证明方发过来的证明 23](#_Toc6884)

[（六） 编译运行 24](#_Toc2687)

[三、 实验结论及心得体会 30](#_Toc22282)

1. **实验要求**

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：x3 + x+ 5 =out。其中out是大家都知道的一个数，这里假设out为 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。

1. **实验过程**
2. **安装libsnark框架**

libsnark是用于开发zkSNARK应用的C++代码库，由SCIPR Lab开发并采用商业友好的MIT许可证（但附有例外条款）在GitHub上（https://github.com/scipr-lab/libsnark）开源。

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。

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

口 Verify∶通过验证密钥验证证明是否正确。

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。

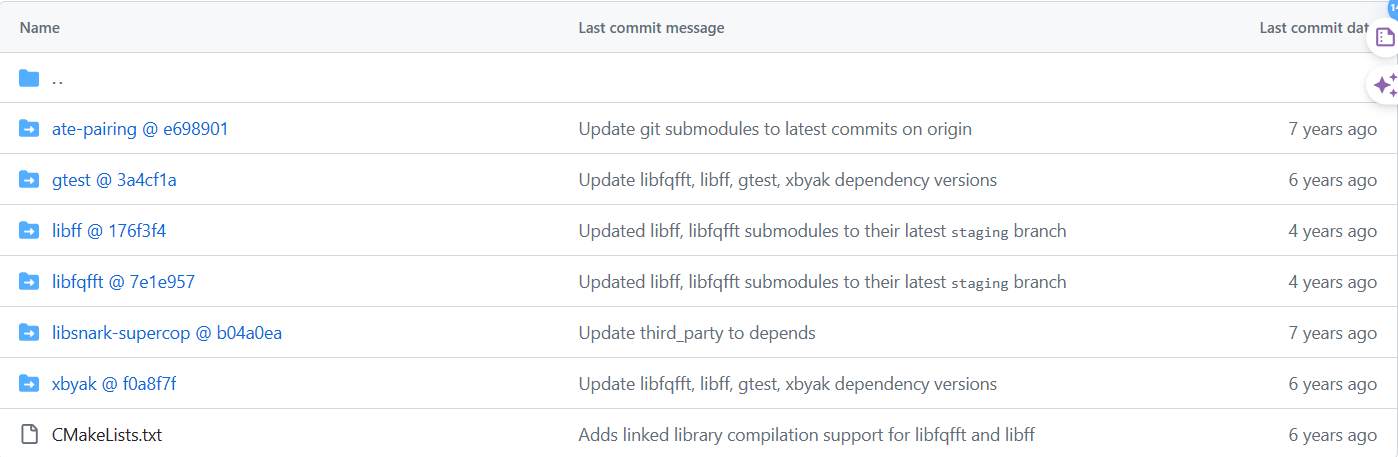
Libsnark安装相对麻烦，它的多个子模块也需要编译安装。

1) 创建名为Libsnark的文件夹

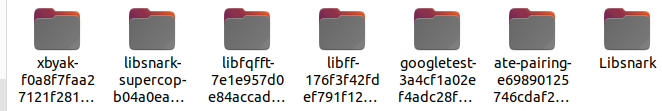
2) 打开<https://github.com/sec-bit/libsnark_abc>，点击“Code”、“Download ZIP”，下载后解压到Libsnark文件夹，得到~/Libsnark/libsnark\_abc-master

3) 打开<https://github.com/scipr-lab/libsnark>，点击“Code”、“Download ZIP”，下载解压后，将其中文件复制到~/Libsnark/libsnark\_abc-master/depends/libsnark文件夹内

4) 打开<https://github.com/scipr-lab/libsnark>，点击“depends”，可以看到六个子模块的链接地址。



分别点击这六个链接并下载解压，得到如下六个文件夹，为方便下文表述，分别将这六个文件夹命名为Libfqfft、Libff、Gtest、Xbyak、Ate-pairing、Libsnark-supercop。



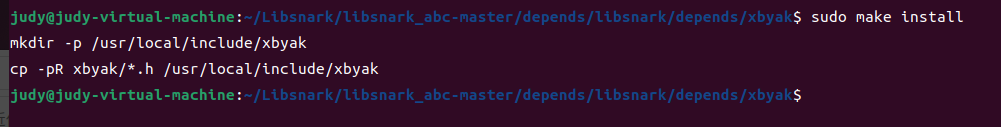
在Linux系统中执行以下命令

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

1. 安装子模块xbyak

将下载得到的文件夹Xbyak内的文件复制到~/Libsnark/libsnark\_abc-master/depends/libsnark/depends/xbyak，并在该目录下打开终端，执行以下命令

|  |
| --- |
| sudo make install |

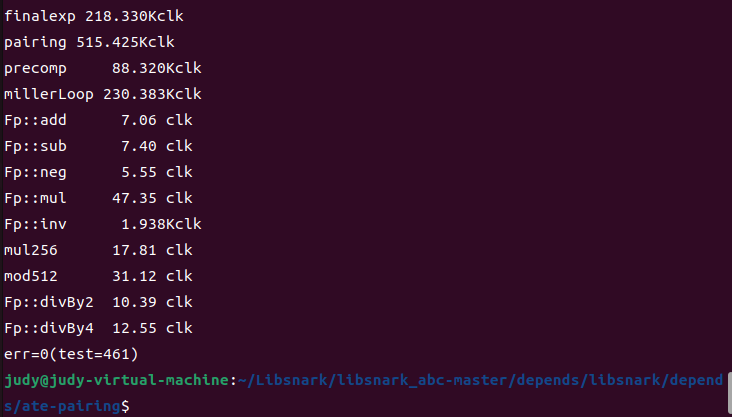


7) 安装子模块ate-pairing

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 |

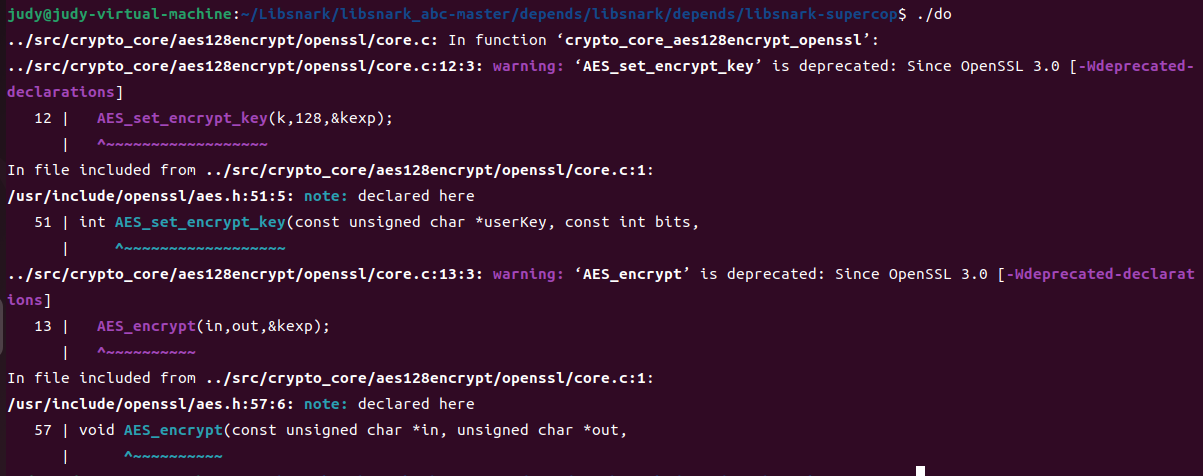
执行test/bn后几行如下



8) 安装子模块libsnark-supercop

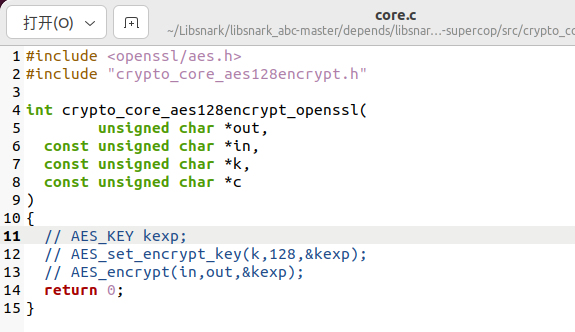
将下载得到的文件夹Libsnark-supercop内的文件复制到~/Libsnark/libsnark\_abc-master/depends/libsnark/depends/libsnark-supercop，并在该目录下打开终端，执行以下命令

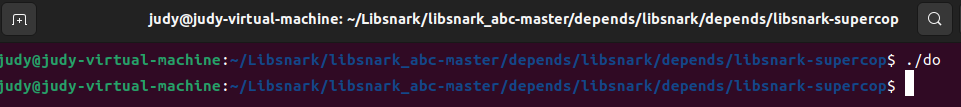
|  |
| --- |
| ./do |



查询资料可知，openssl进入3.0版本，openssl的加解密代码也出现了一些变化，编译时会有如下错误：Since OpenSSL 3.0。如果使用OpenSSL 1.1.1 sdk编译则没有上述错误，使用3.0以上的openssl sdk就会报错，因为3.0的不兼容1.0的sdk。如果你想继续使用已弃用的函数，并且不想更改代码，可以考虑禁用特定的编译警告。在 Visual Studio 中，可以使用 #pragma warning(disable: 4996) 来禁用这个特定的警告。

将core.c文件中重定义的两个函数注释掉，不再出现警告





9) 安装子模块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

10) 安装子模块libff

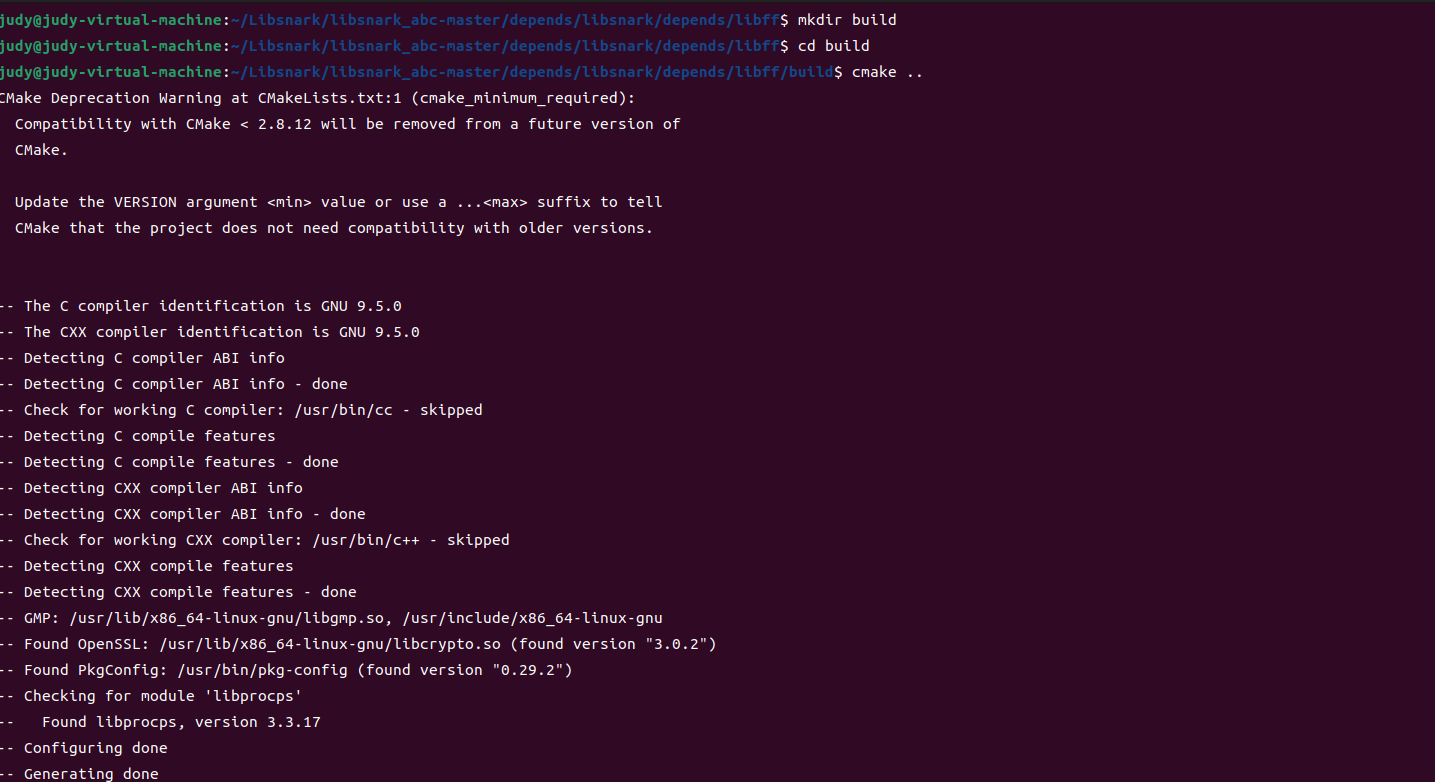
将下载得到的文件夹Libff内的文件复制到~/Libsnark/libsnark\_abc-master/depends/libsnark/depends/libff。点击libff->depends，可以看到一个ate-pairing文件夹和一个xbyak文件夹，这是libff需要的依赖项。打开这两个文件夹，会发现它们是空的，这时候需要将下载得到的Ate-pairing和Xbyak内的文件复制到这两个文件夹下。

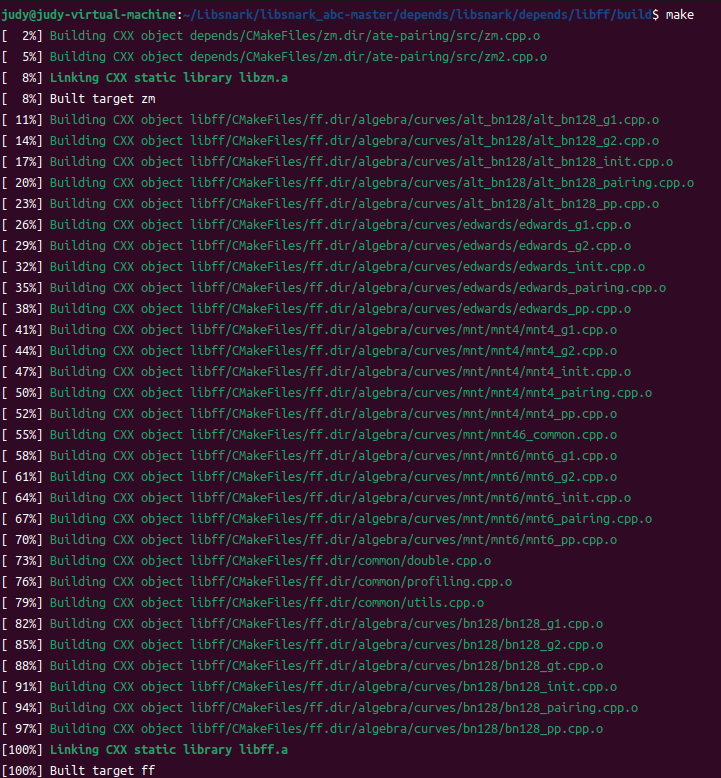
在~/Libsnark/libsnark\_abc-master/depends/libsnark/depends/libff下打开终端，执行命令:

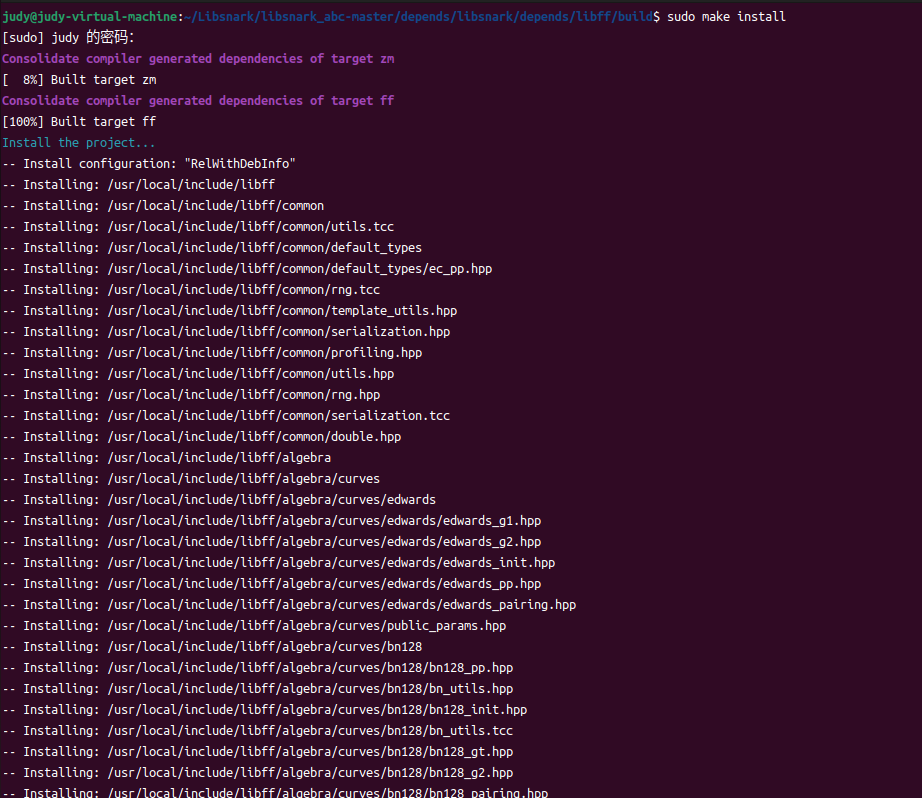
|  |
| --- |
| mkdir build  cd build  cmake ..  make  sudo make install |

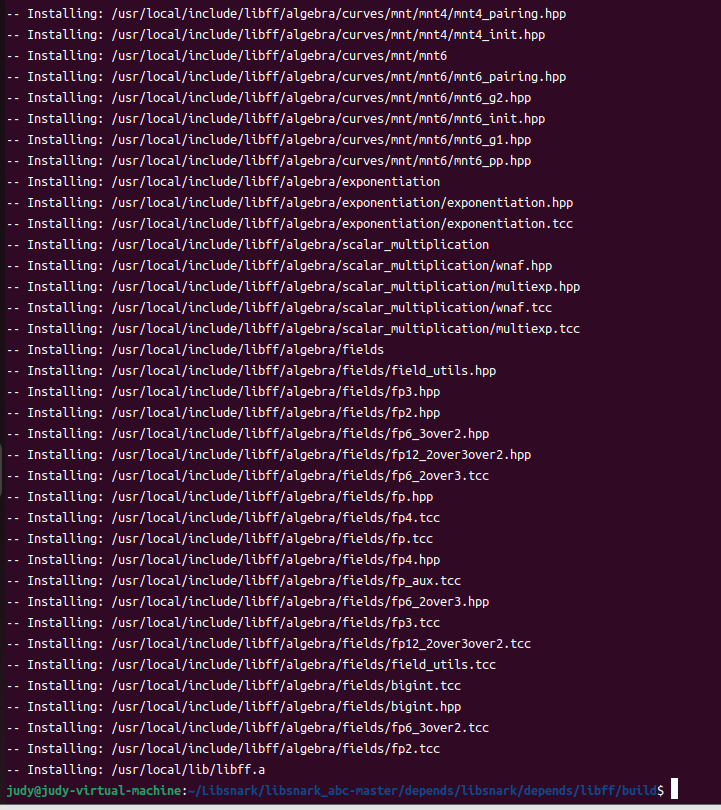
安装完之后检测是否安装成功，执行以下命令

|  |
| --- |
| make check |

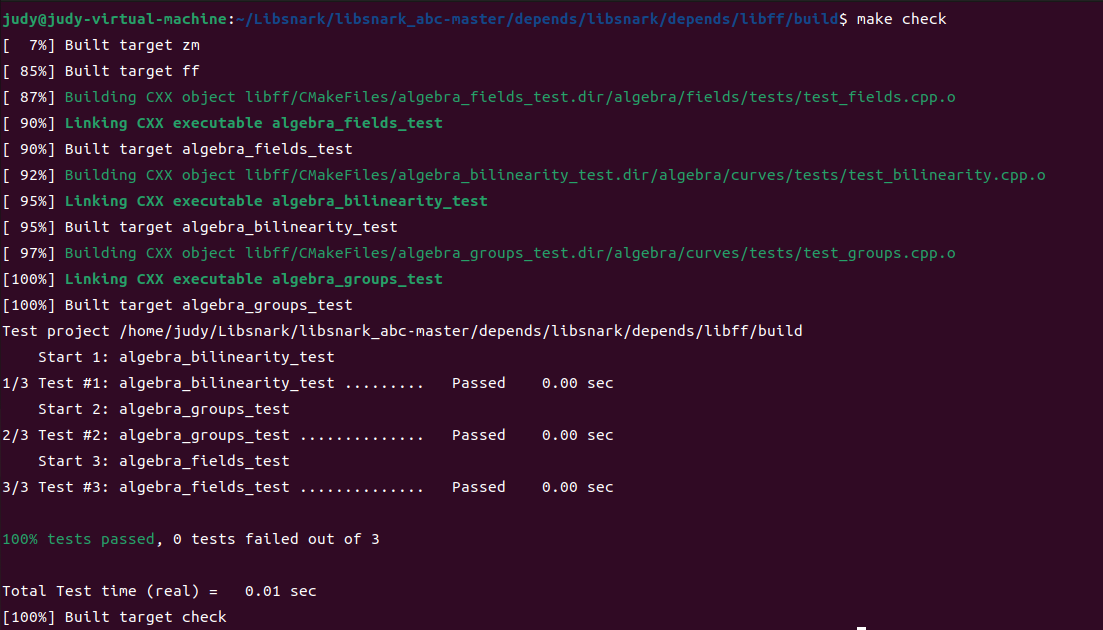








安装完检测安装是否成功，如下所示表明已安装成功：



11) 安装子模块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。

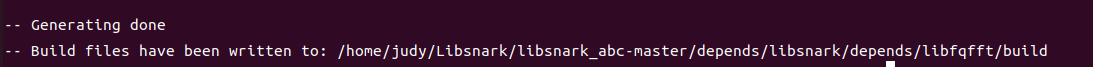
注意libff里还有depends文件夹，里面的ate-pairing和xbyaky也是空的，需要将下载得到的airing和Xbyak文件夹内的文件复制进去。

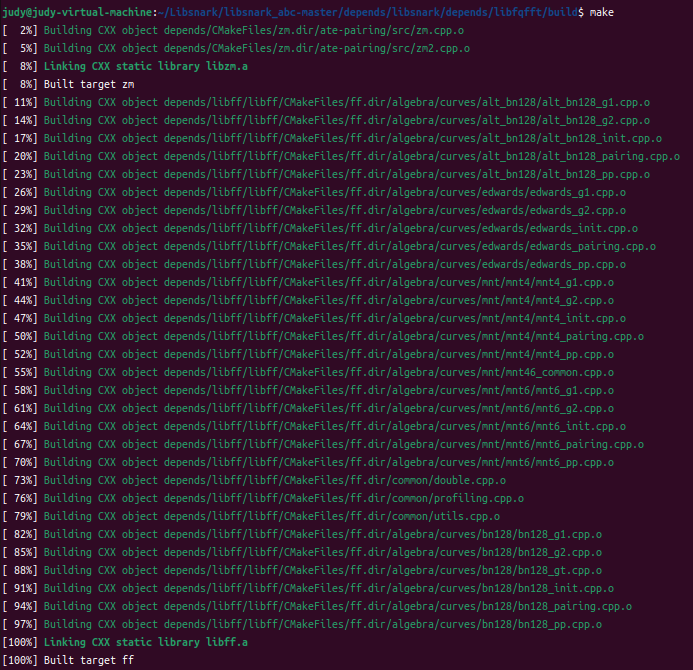
在~/Libsnark/libsnark\_abc-master/depends/libsnark/depends/libfqfft下打开终端，执行命令：

|  |
| --- |
| mkdir build  cd build  cmake ..  make  sudo make install |

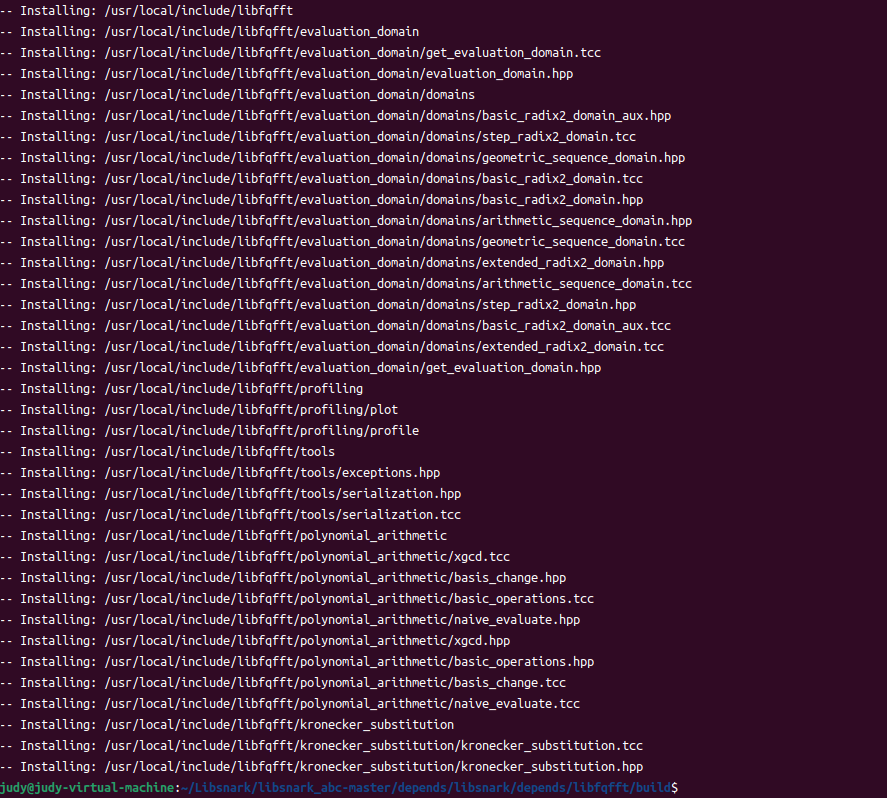
安装完之后检测是否安装成功，执行以下命令

|  |
| --- |
| make check |

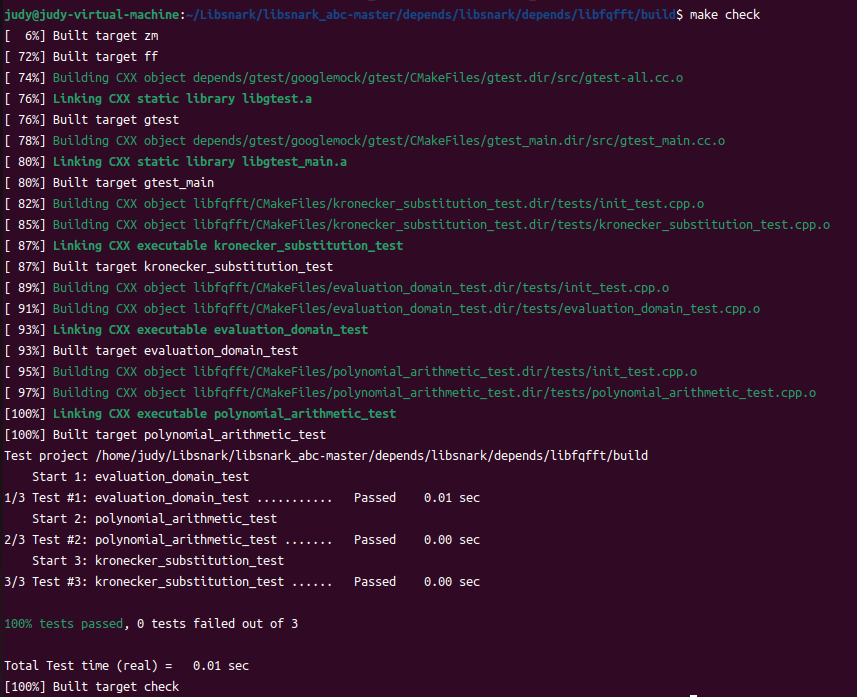








安装完检测安装是否成功，如下所示表明已安装成功：



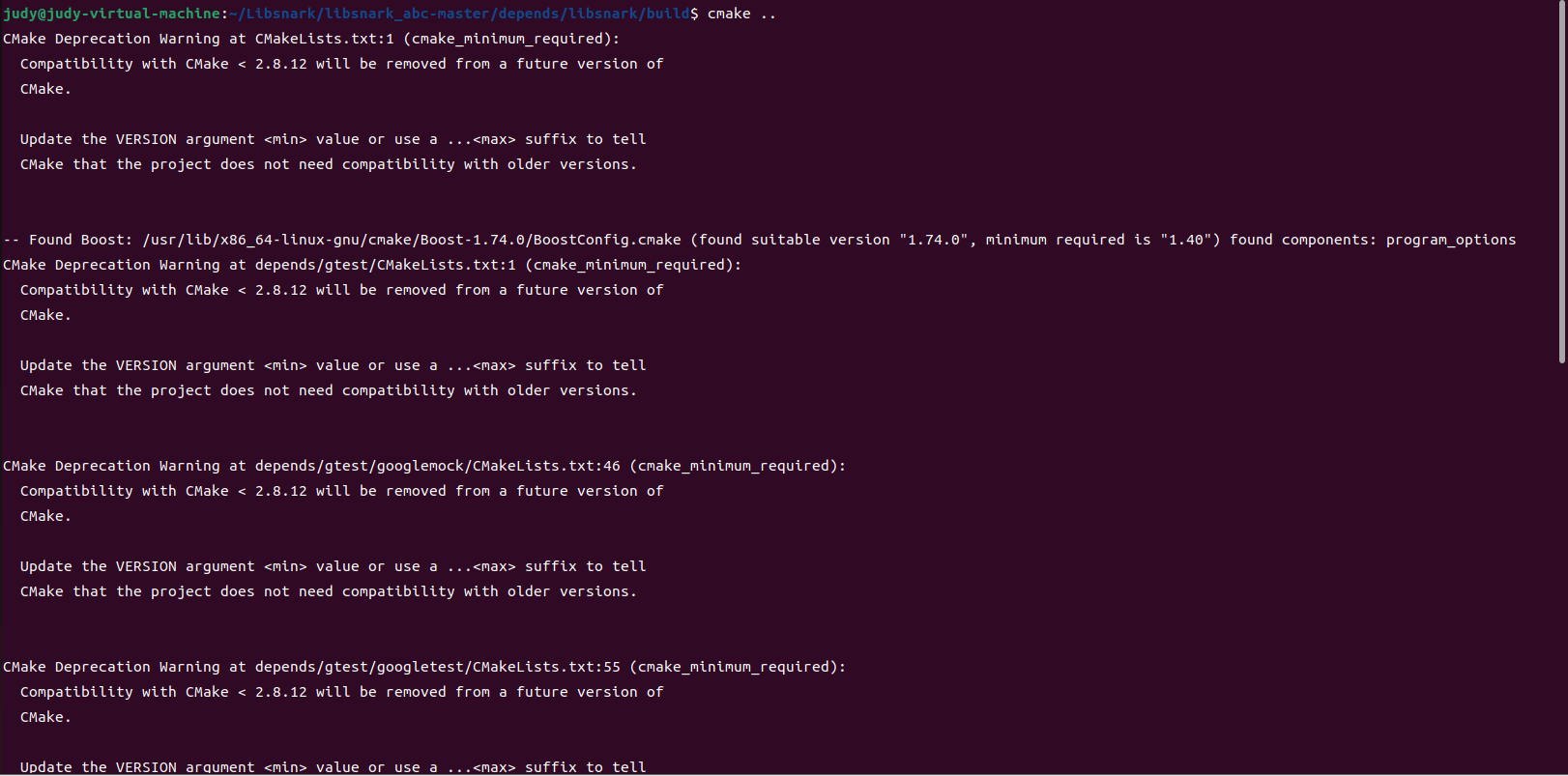
12) libsnark编译安装

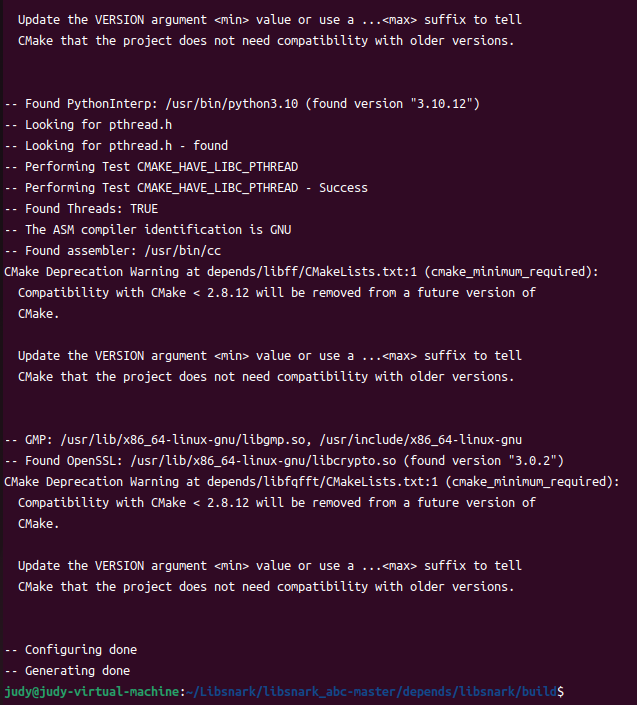
在~/Libsnark/libsnark\_abc-master/depends/libsnark下打开终端，执行以下命令：

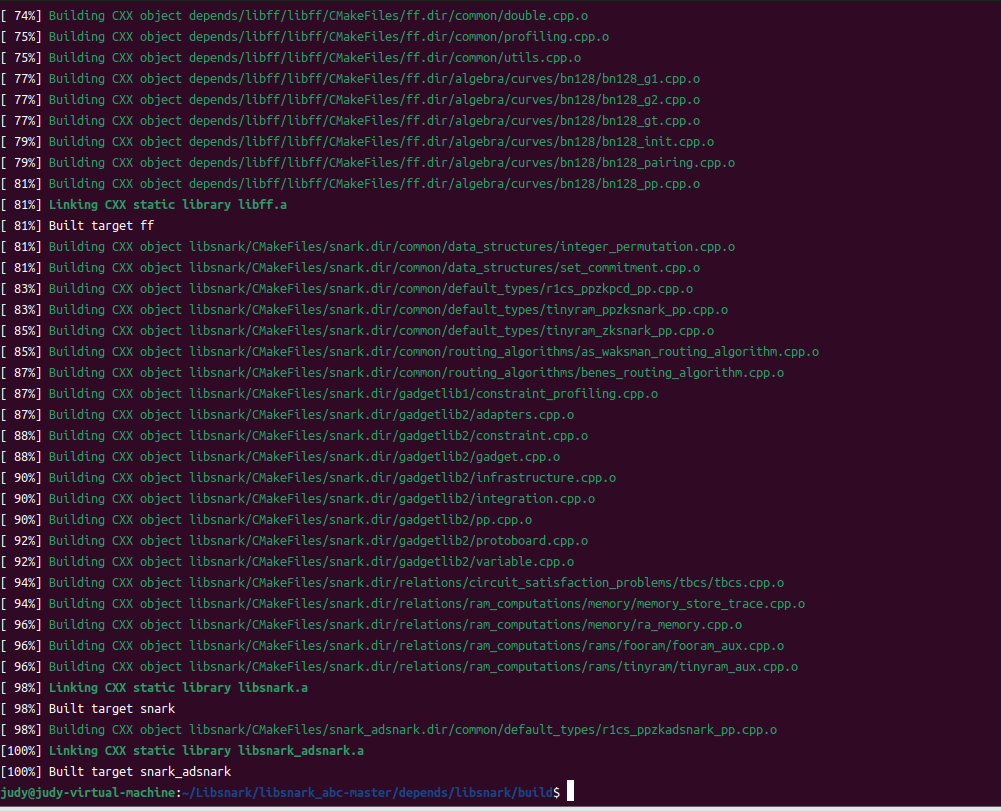
|  |
| --- |
| mkdir build  cd build  cmake ..  make |

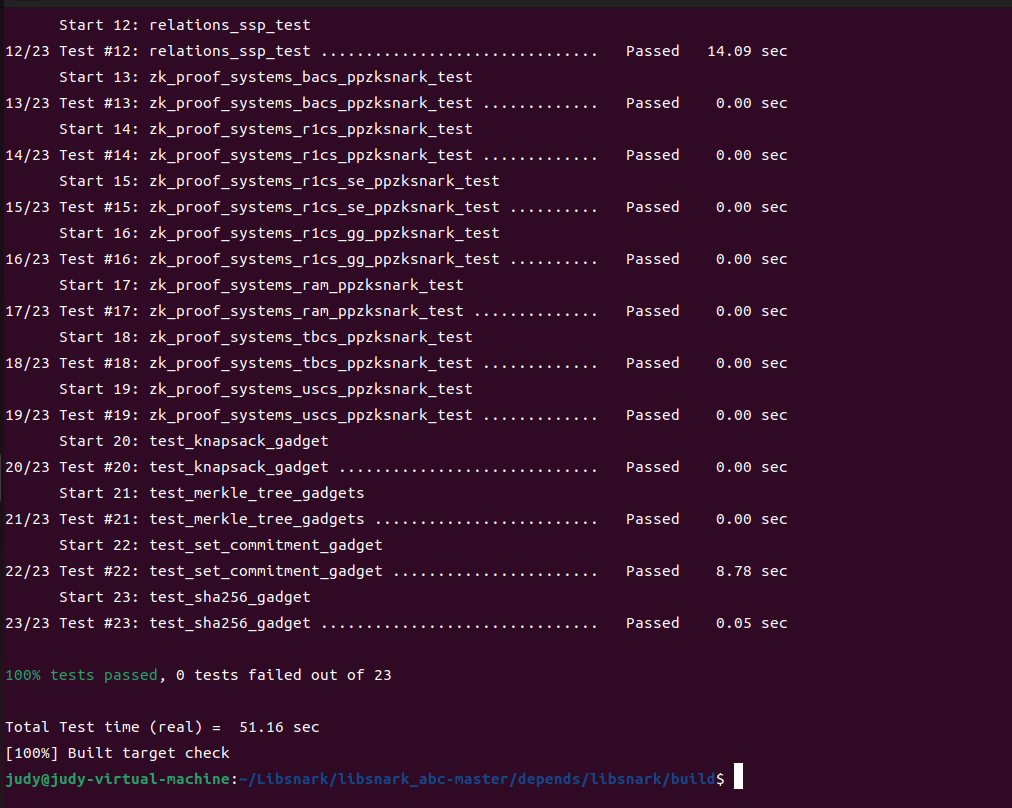
安装完之后检测是否安装成功，执行以下命令

|  |
| --- |
| make check |





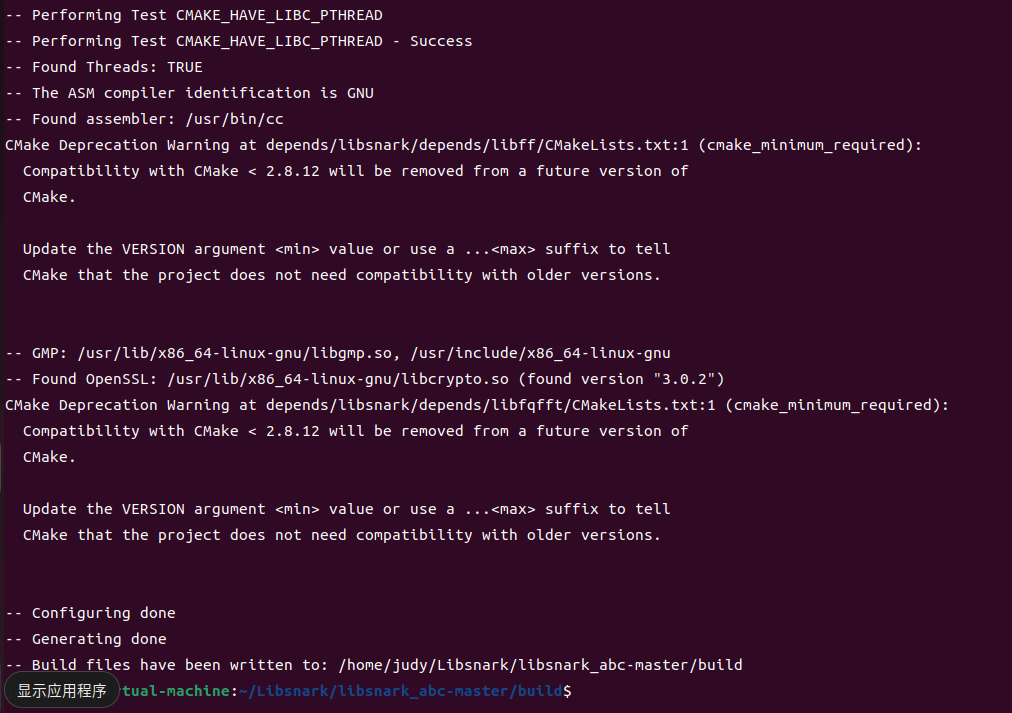


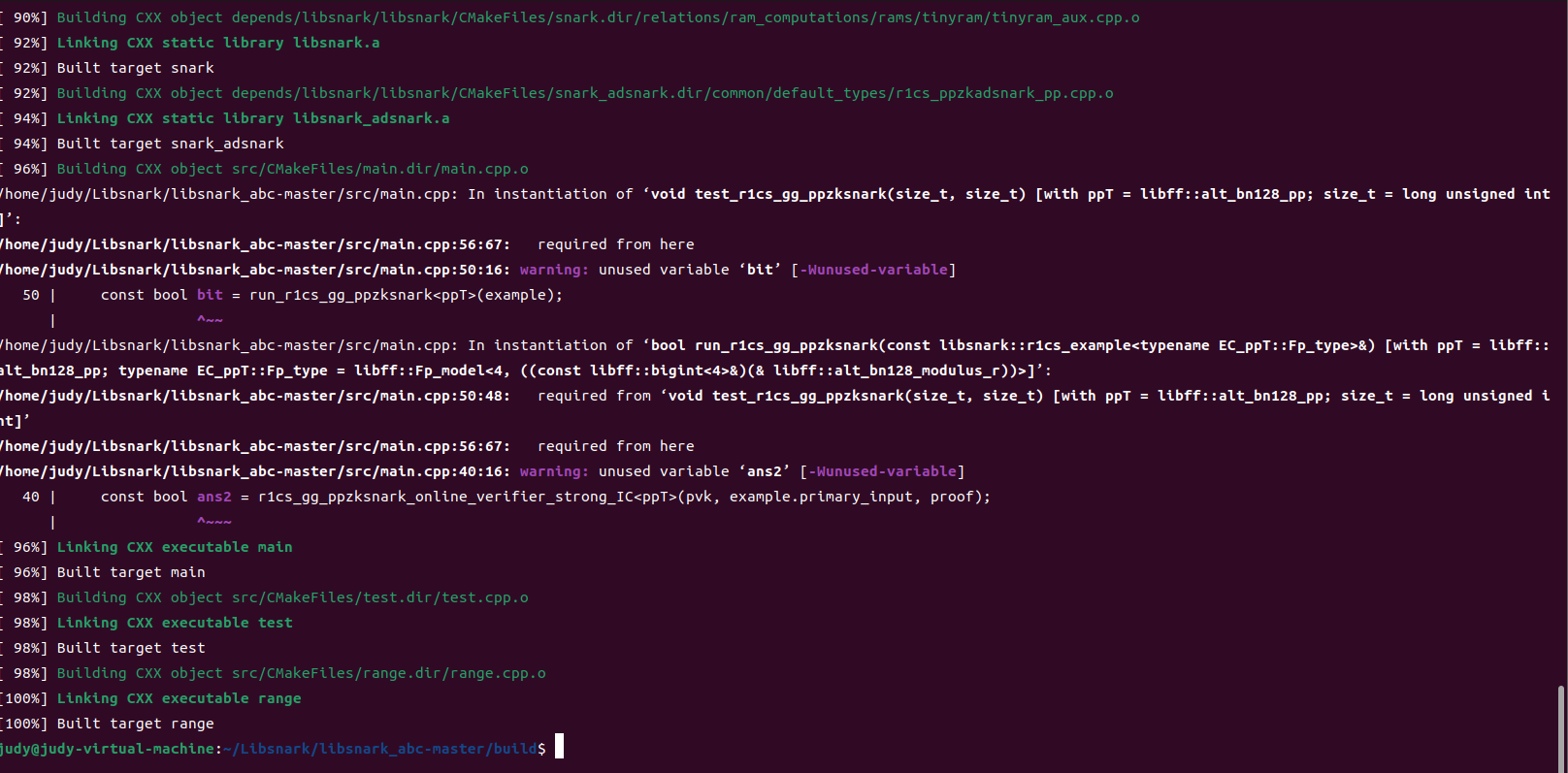


13) 整体编译安装

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

|  |
| --- |
| mkdir build  cd build  cmake ..  make |



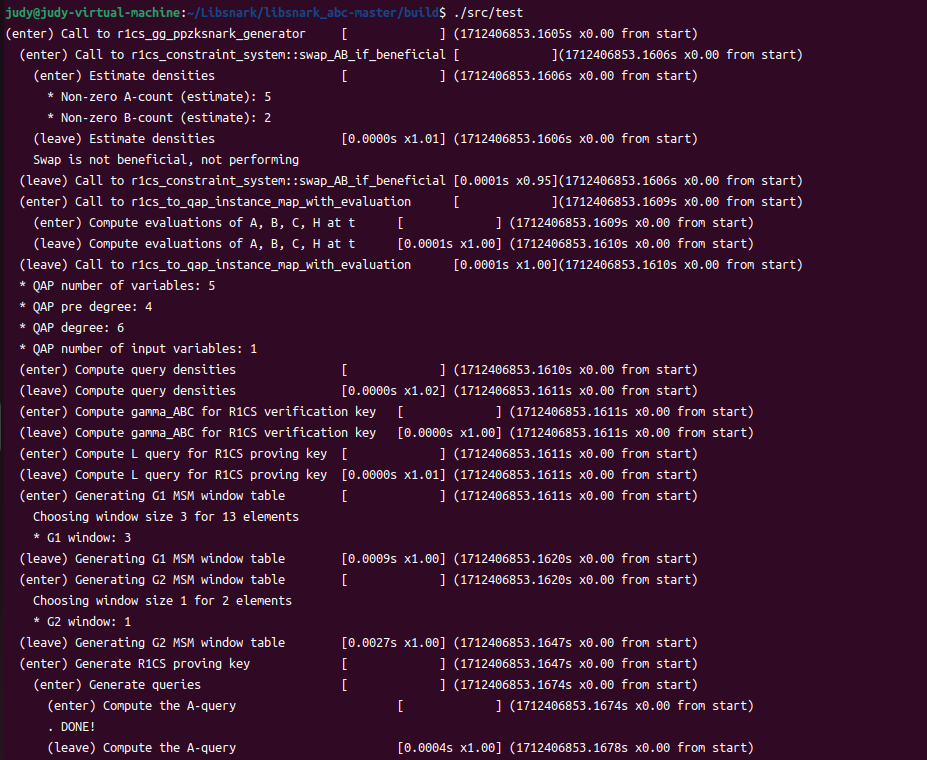


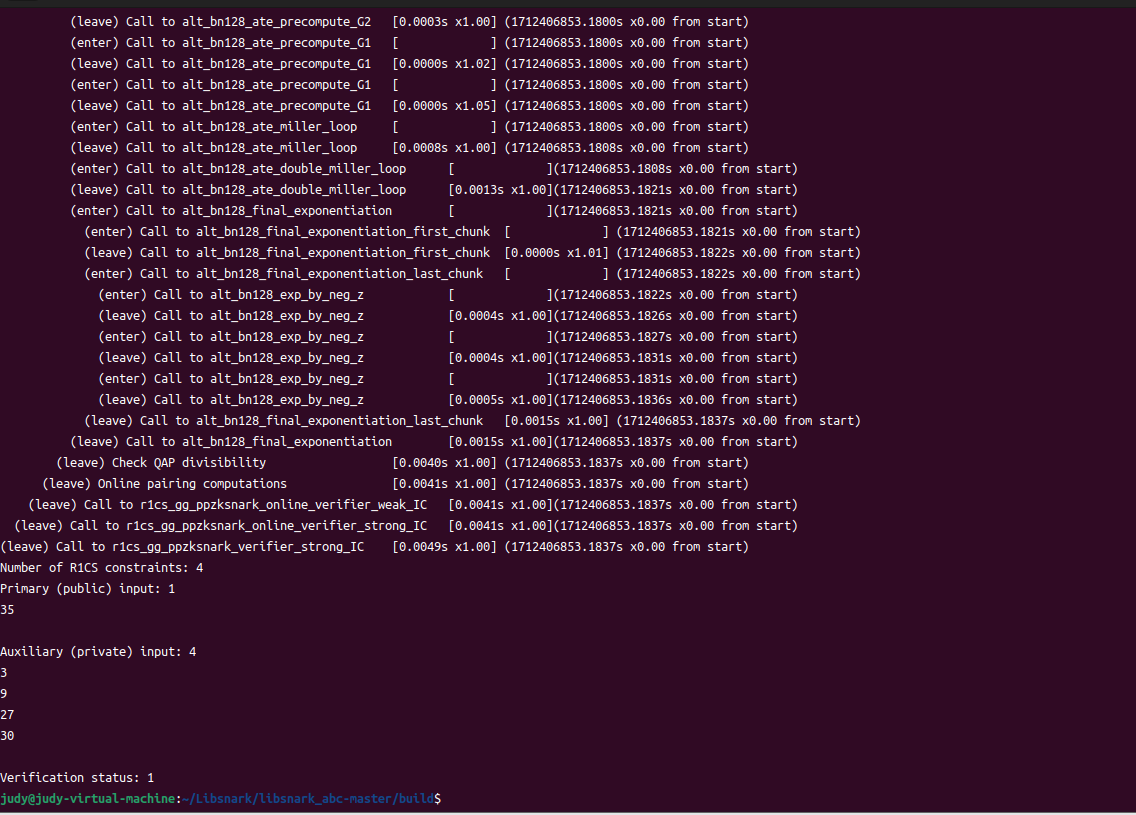
14) 运行代码

执行以下命令：

|  |
| --- |
| ./src/test |

最终出现如下日志，则说明已顺利拥有了zkSNARK应用开发环境，并成功跑了第一个zk-SNARKs的demo。





1. **将待证明的命题表达为R1CS**
2. 用R1CS描述电路

将算数电路拍平成多个或者形式的等式，其中可以是加、减、乘、除运算符中的一种。对于x3 + x+ 5 =out，可以拍平成如下几个等式：

|  |
| --- |
|  |

1. 使用原型版搭建电路

使用原型板protoboard搭建电路。在电气工程中，原型板（Protoboard）是用于连接电路和芯片的。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。

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。

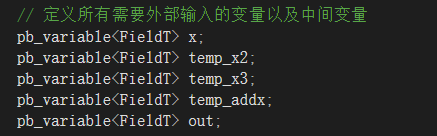
仿照教材实验的common.hpp代码，修改本实验中的Common.hpp如下：

|  |
| --- |
| // 代码开头引用了三个头文件：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；pb\_variable 则是用来定义电 路相关的变量。  #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;  // 定义使用的有限域  typedef libff::Fr<default\_r1cs\_gg\_ppzksnark\_pp> FieldT;  // 定义创建面包板的函数  protoboard<FieldT> build\_protoboard(int \*secret)  {  // 初始化曲线参数  default\_r1cs\_gg\_ppzksnark\_pp::init\_public\_params();  // 创建面包板  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> w\_4;  // pb\_variable<FieldT> w\_5;  // pb\_variable<FieldT> out;    // 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> temp\_x2;  pb\_variable<FieldT> temp\_x3;  pb\_variable<FieldT> temp\_addx;  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。  // 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");  // w\_4.allocate(pb, "w\_4");  // w\_5.allocate(pb, "w\_5");  // 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。  out.allocate(pb, "out");  x.allocate(pb, "x");  temp\_x2.allocate(pb, "temp\_x2");  temp\_x3.allocate(pb, "temp\_x3");  temp\_addx.allocate(pb, "temp\_addx");  // 定义公有的变量的数量，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（前面 out 是公开的）。  pb.set\_input\_sizes(1);  // 为公有变量赋值  // pb.val(out) = 0;  pb.val(out) = 35;  // 至此，所有变量都已经顺利与 protoboard 相连，下面需要确定的是这些变量间的约束关系。  // 如下调用protoboard 的add\_r1cs\_constraint()函数，为pb添加形如a \* b = c的r1cs\_constraint。即r1cs\_constraint<FieldT>(a, b, c)中参数应该满足a \* b = c。  // 根据注释不难理解每个等式和约束之间的关系。  // x-1= w\_1  // pb.add\_r1cs\_constraint(r1cs\_constraint<FieldT>(x - 1, 1, w\_1));  // x-2= w\_2  // pb.add\_r1cs\_constraint(r1cs\_constraint<FieldT>(x - 2, 1, w\_2));  // x-3= w\_3  // pb.add\_r1cs\_constraint(r1cs\_constraint<FieldT>(x - 3, 1, w\_3));  // x\*w\_1=w\_4  // pb.add\_r1cs\_constraint(r1cs\_constraint<FieldT>(x, w\_1, w\_4));  // w\_2\*w\_4=w\_5  // pb.add\_r1cs\_constraint(r1cs\_constraint<FieldT>(w\_2, w\_4, w\_5));  // w\_3\*w\_5=out  // pb.add\_r1cs\_constraint(r1cs\_constraint<FieldT>(w\_3, w\_5, out));    // 至此，所有变量都已经顺利与 protoboard 相连，下面需要确定的是这些变量间的约束关系。  // 如下调用protoboard 的add\_r1cs\_constraint()函数，为pb添加形如a \* b = c的r1cs\_constraint。即r1cs\_constraint<FieldT>(a, b, c)中参数应该满足a \* b = c。  // 根据注释不难理解每个等式和约束之间的关系。  // x\*x = temp\_x2  pb.add\_r1cs\_constraint(r1cs\_constraint<FieldT>(x, x, temp\_x2));  // temp\_x2 \* x = temp\_x3  pb.add\_r1cs\_constraint(r1cs\_constraint<FieldT>(temp\_x2, x, temp\_x3));  // temp\_x3 + x = temp\_addx  pb.add\_r1cs\_constraint(r1cs\_constraint<FieldT>(temp\_x3 + x, 1, temp\_addx));  // temp\_addx + 5 = out  pb.add\_r1cs\_constraint(r1cs\_constraint<FieldT>(temp\_addx + 5, 1, out));  //证明者在生成证明阶段传入私密输入，为私密变量赋值，其他阶段为NULL  // 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];  pb.val(w\_4) = secret[4];  pb.val(w\_5) = secret[5];  }\*/    // 证明者在生成证明阶段传入私密输入，为私密变量赋值，其他阶段为 NULL  if (secret != NULL)  {  pb.val(x) = secret[0];  pb.val(temp\_x2) = secret[1];  pb.val(temp\_x3) = secret[2];  pb.val(temp\_addx) = secret[3];  }  return pb;  } |

主要修改的地方在于，R1CS描述电路需要将算数电路拍平成多个或者形式的等式，本实验中需要拍平成如下几个等式，需要根据如下的等式对代码的相关部分进行修改。

|  |
| --- |
|  |

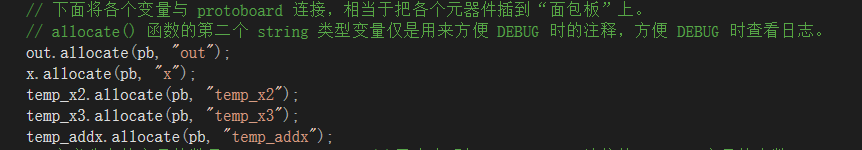




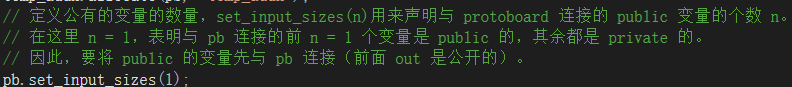
定义protoboard和变量，其中变量1会在protoboard中自动定义。在pb中声明变量，将各个变量与pb关联起来，以便在pb中使用变量定义R1CS约束；

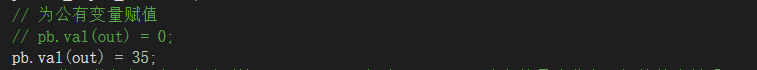
pb中primary为public变量，auxiliary为private变量，primary变量在前面定义（如“out”）。

x、temp\_x2、temp\_x3、temp\_addx是电路的中间变量，out为输出，公有变量。temp\_x2=x\*x，temp\_x3=temp\_x2\*x，即x^3，temp\_addx=temp\_x3+x，即x^3+x。Out=temp\_addx+5，即out=x^3+x+5。



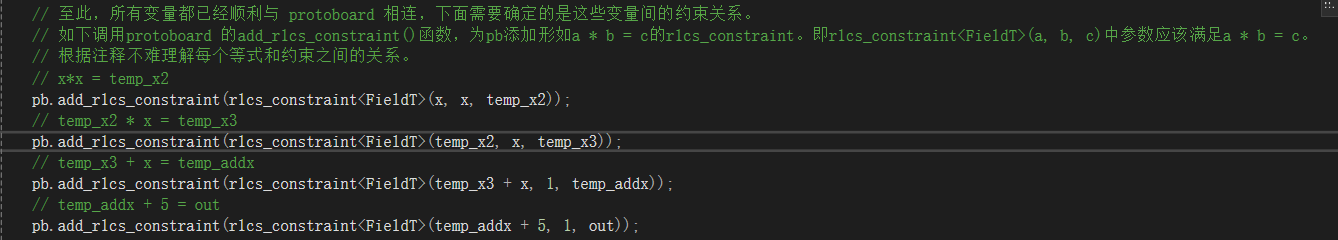
使用set\_input\_sizes(n)函数指出前面的n个变量为primary变量，后面剩下的则为auxiliary变量；第一个为primary变量，后面的都是auxiliary变量。

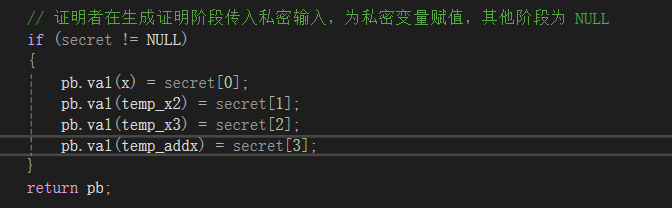




公有变量out赋值为35。

根据4个等式编写如下代码





如果 secret 不为 NULL，说明当前处于生成证明的阶段，此时通过 pb.val(x) = secret[0] 的方式为变量 x 赋值，pb.val(temp\_x2) = secret[1] 的方式为变量temp\_x2赋值，pb.val(temp\_x3) = secret[2]的方式为temp\_x3赋值，pb.val(temp\_addx) = secret[3] 的方式为temp\_addx赋值。

如果 secret 为 NULL，则说明当前处于验证证明的阶段，此时不需要为私密变量赋值，直接返回 protoboard 即可。

1. **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。其中，证明密钥供证明者使用，验证密钥供验证者使用。

编写代码如下，将这段代码放在mysetup.cpp中。

这里mysetup.cpp的代码与教材参考的实验代码保持一致。

|  |
| --- |
| #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;  } |

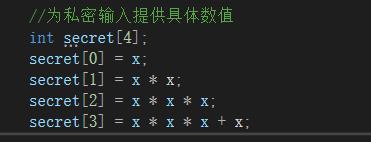
这段代码实现了使用libsnark库构建零知识证明系统的一部分功能。创建了一个protoboard 对象 pb，调用 build\_protoboard() 函数构建了一个约束系统，是零知识证明系统的一部分。该约束系统由用户自定义的约束构成。通过调用 pb.get\_constraint\_system() 函数获取构建的约束系统。使用 r1cs\_gg\_ppzksnark\_generator 函数生成了证明密钥对 keypair，这个密钥对包含了证明密钥 (pk) 和验证密钥 (vk)。将生成的证明密钥 (pk) 和验证密钥 (vk) 分别保存到名为 pk.raw 和 vk.raw 的文件中。

1. **证明方使用证明密钥和其可行解构造证明**

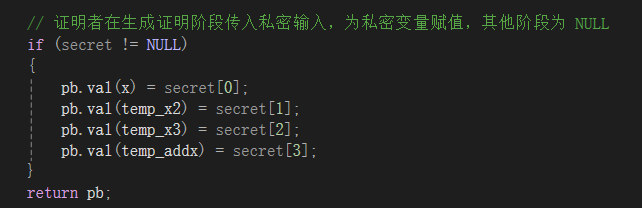
在定义面包板时，已为public input提供具体数值，在构造证明阶段，证明者只需为private input提供具体数值。再把public input以及private input的数值传给prover函数生成证明。生成的证明保存到proof.raw文件中供验证者使用。编写代码如下，将这段代码放在myprove.cpp中。

|  |
| --- |
| #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[6];  // secret[0] = x;  // secret[1] = x - 1;  // secret[2] = x - 2;  // secret[3] = x - 3;  // secret[4] = x \* (x - 1);  // secret[5] = x \* (x - 1) \* (x - 2);    //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;  } |

myprove.cpp需要对本实验要求中要满足的等式，做出如下修改



需要注意的是，这里需要与common.hpp中生成证明的部分保持对应



1. **验证方使用验证密钥验证证明方发过来的证明**

最后使用verifier函数校验证明。如果verified = 1则说明证明验证成功。编写代码如下，将这段代码放在myverify.cpp中。

这里的myverify.cpp与教材实验保持一致即可。

|  |
| --- |
| #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;  } |

待证明问题

算数电路C(v,w)

Setup(C)

Prove(pk,v,w)

Prove(vk,v,proof)

输出0/1

pk

vk

proof

1. **编译运行**

在~/Libsnark/libsnark\_abc-master/src下打开终端，输入如下命令，创建common.hpp、mysetup.cpp、myprove.cpp、myverify.cpp文件

|  |
| --- |
| touch common.hpp  touch mysetup.cpp  touch myprove.cpp  touch myverify.cpp |

分别打开common.hpp、mysetup.cpp、myprove.cpp、myverify.cpp，将对应代码复制进文件中。

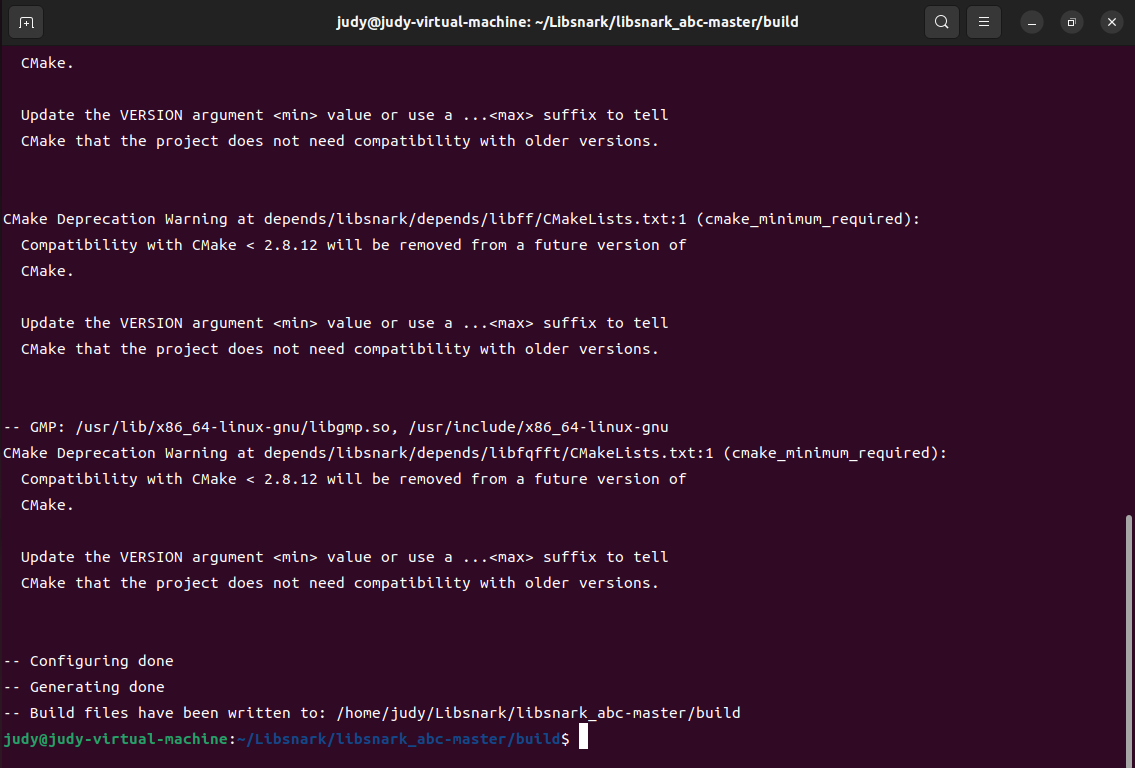
打开~/Libsnark/libsnark\_abc-master/src目录下的CMakeLists.txt文件，将如下代码复制到文件末尾。

|  |
| --- |
| 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  ) |

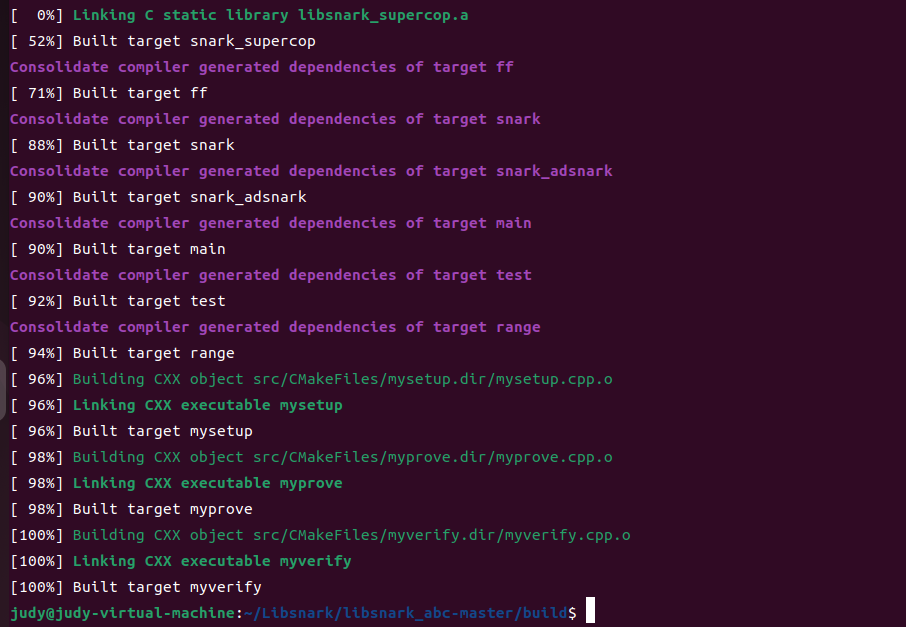
在~/Libsnark/libsnark\_abc-master/build下打开终端，执行以下命令：

|  |
| --- |
| cmake ..  make  cd src  ./mysetup  ./myprove  3  ./myverify |

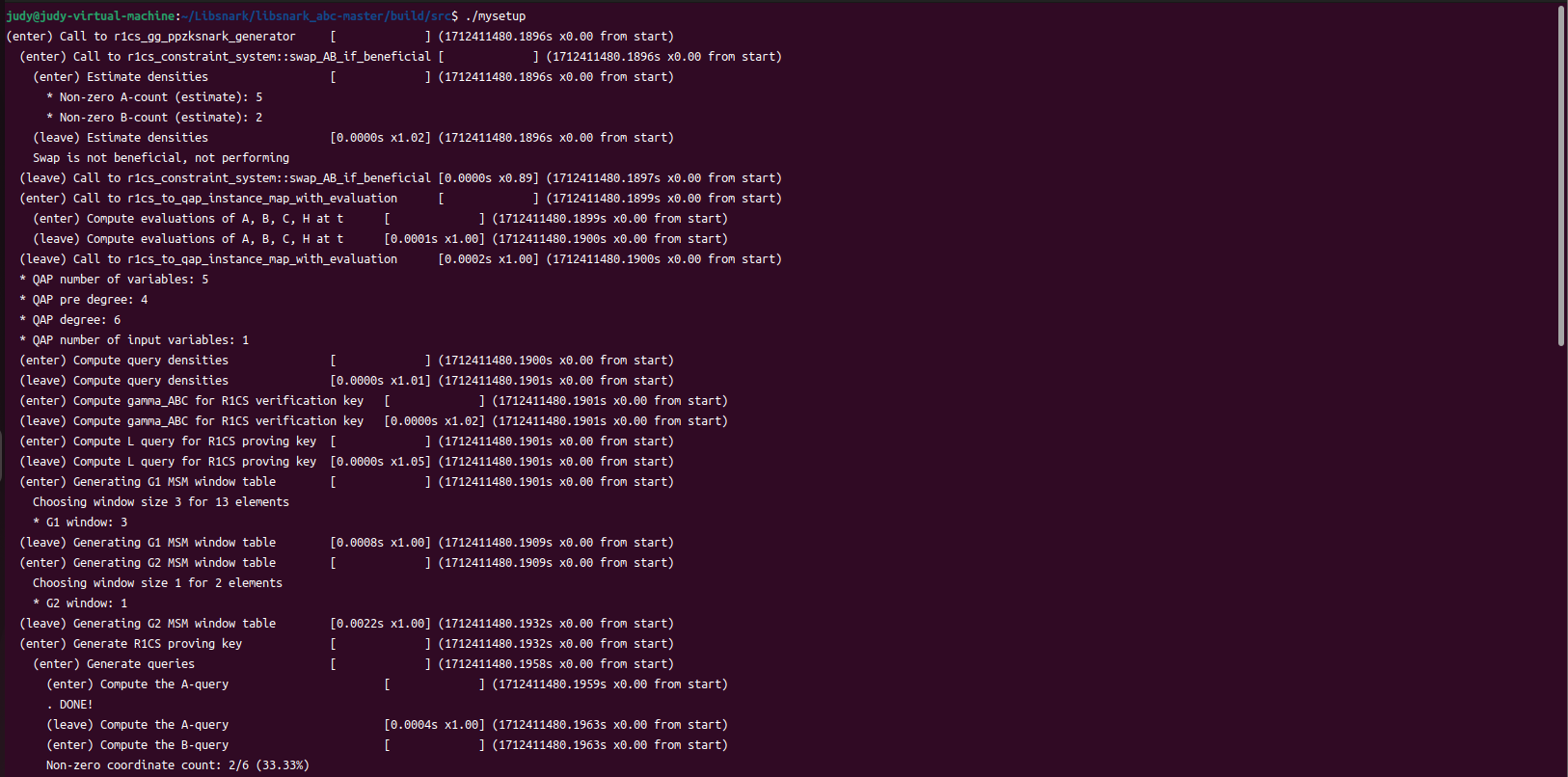
|  |
| --- |
| cmake .. |

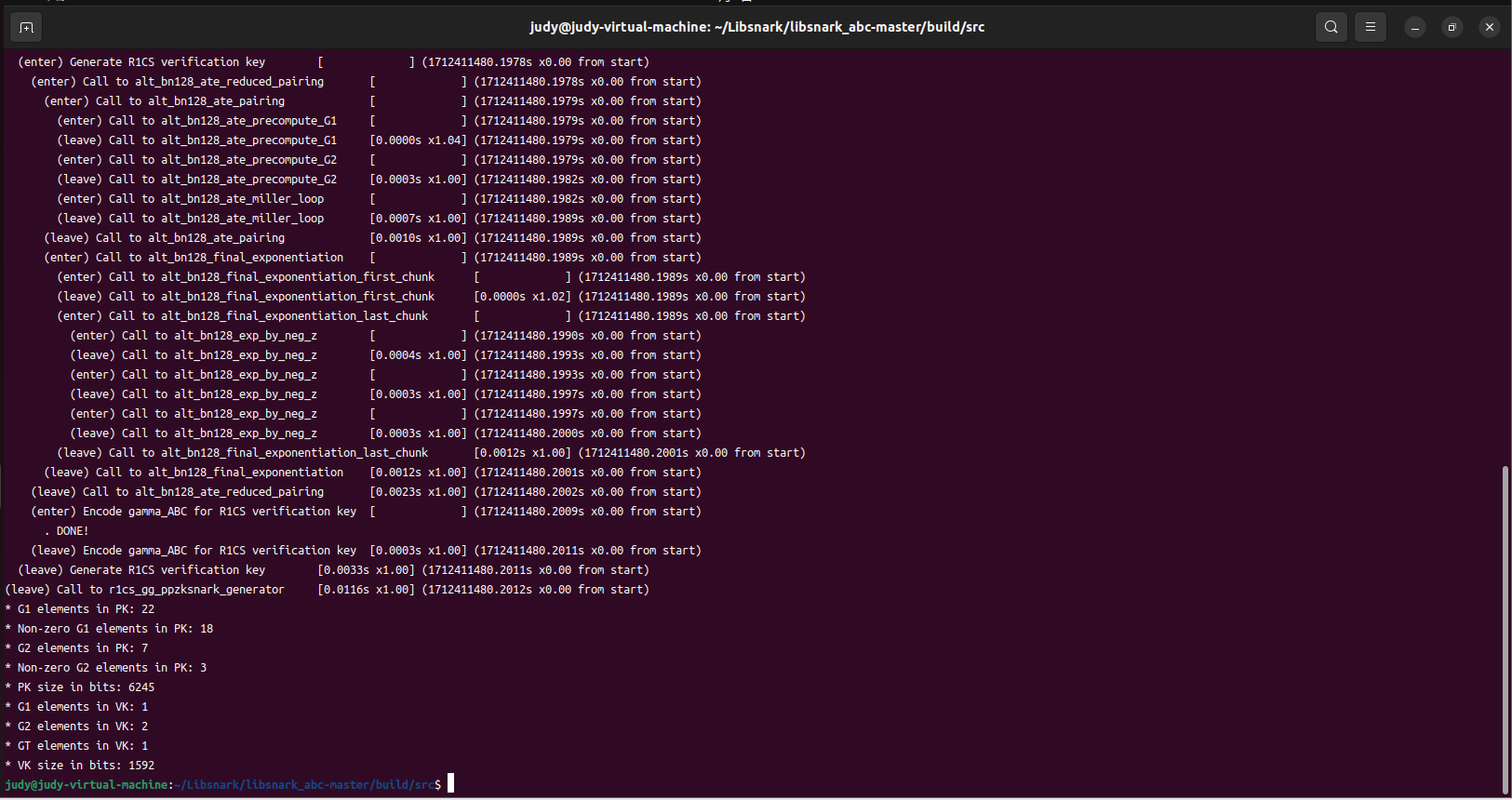


|  |
| --- |
| make |



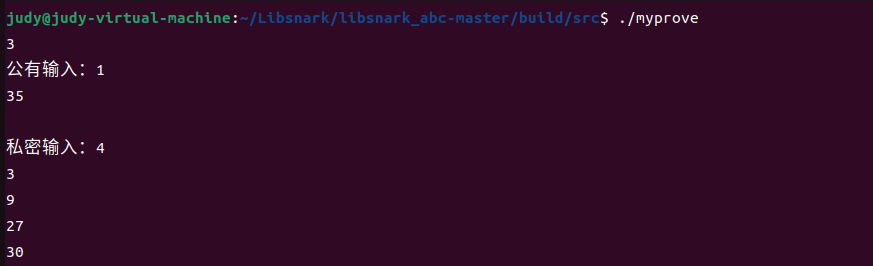
|  |
| --- |
| cd src  ./mysetup |

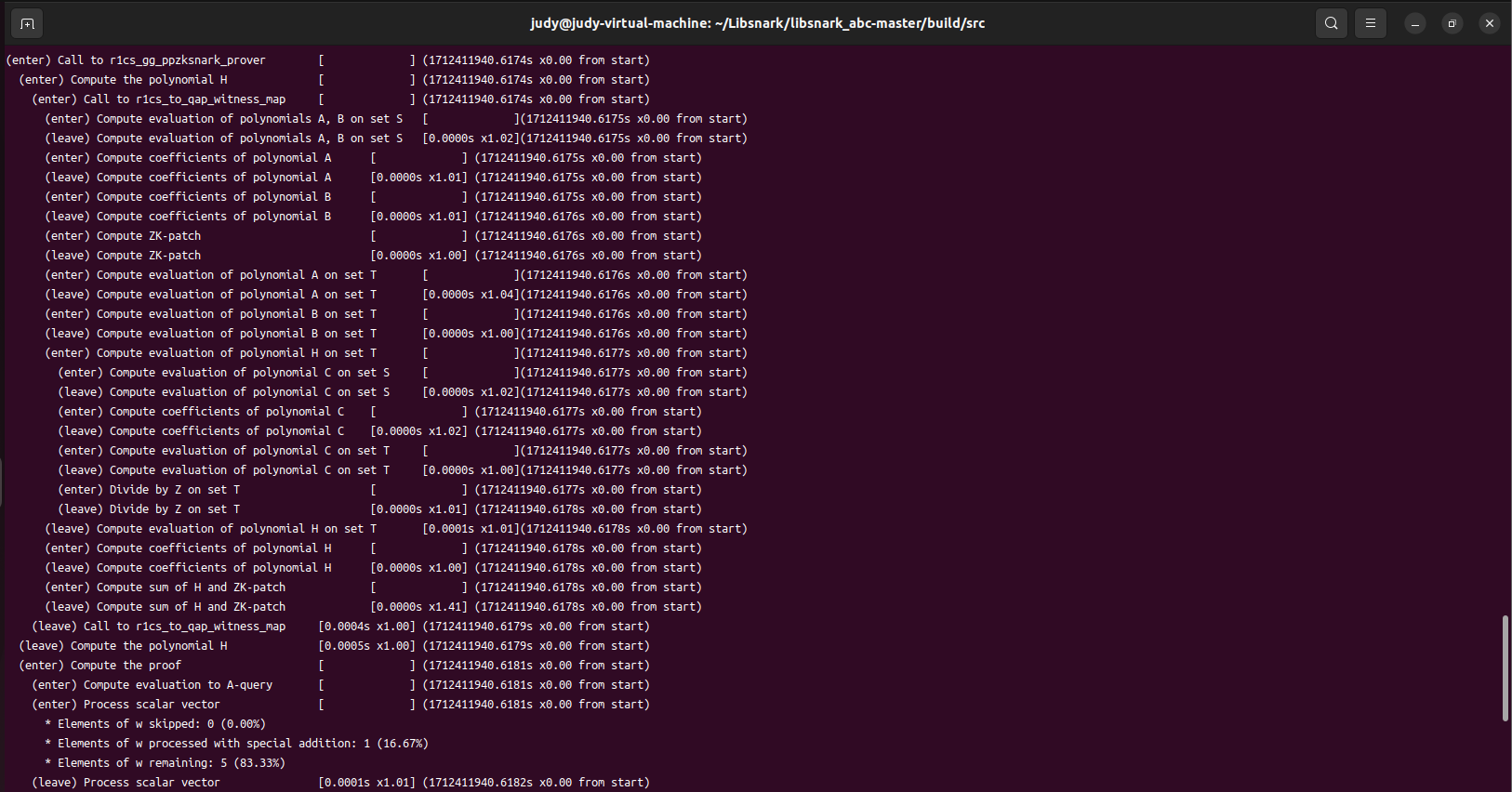


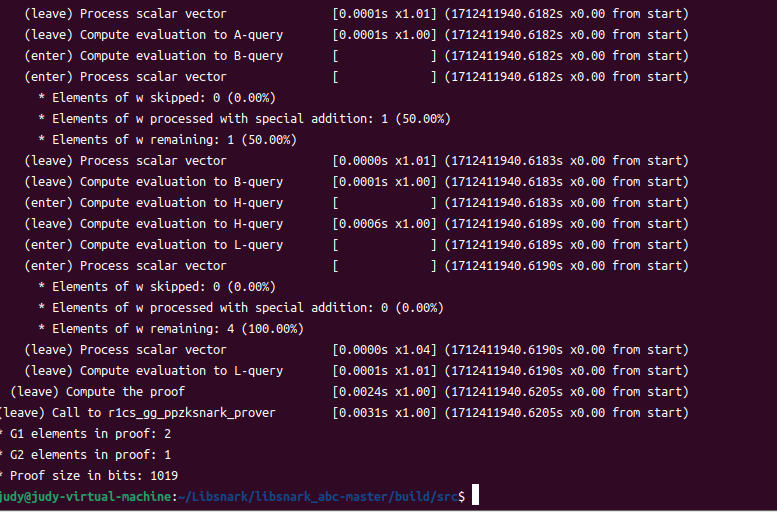


|  |
| --- |
| ./myprove  3 |

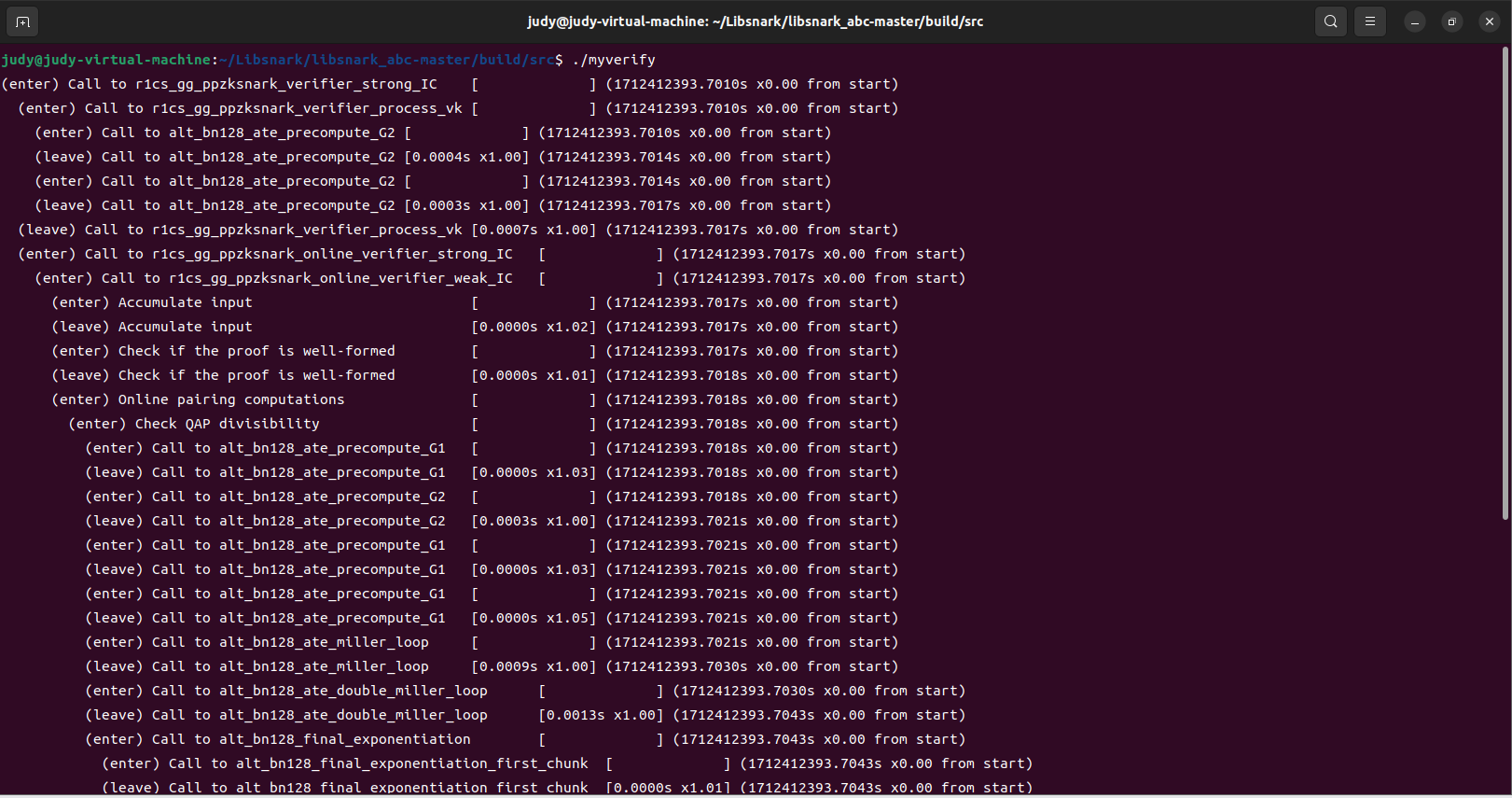
输入x=3之后，公有输入1，私密输入4。







|  |
| --- |
| ./myverify |



运行结果如下：验证结果为1，说明通过证明的验证



1. **实验结论及心得体会**

本次实验是针对零知识证明的实践，通过搭建电路、生成证明密钥和验证密钥、生成证明、以及验证证明的全流程，理解了零知识证明的原理与实现过程。

首先，在实验中我了解到了如何使用libsnark库构建零知识证明系统的各个组成部分，包括protoboard的构建、约束系统的定义、证明密钥和验证密钥的生成等。这一过程中需要理解电路的构建以及R1CS的表示方法，使用了r1cs\_gg\_ppzksnark算法生成零知识证明。

其次，在实验中我对零知识证明的工作流程有了更清晰的认识，即初始设置阶段生成公钥、生成证明阶段根据私密输入生成证明、验证阶段使用验证密钥验证证明。每个阶段都有其独特的任务和实现方式，通过实际操作深入理解了这一过程。

最后，在实验中我也遇到了一些挑战，如libsnark库的安装与配置、代码的编写与调试等。通过查阅资料、仔细阅读文档以及反复尝试，逐渐解决了这些问题，增强了自己的实践能力和解决问题的能力。

总的来说，本次实验让我深入了解了零知识证明的原理与实现方法，同时也提升了我的编程技能和问题解决能力。这对我今后的学习和研究都有着积极的影响，是一次十分有价值的实践经验。