

南开大学

数据安全 课程实验报告

零知识证明实践



学院 网络空间安全学院

专业 信息安全

姓名 齐明杰

学号 2113997

2024 年 3 月 30 日

目 录

1	实验目的	3
2	实验原理	3
3	实验过程	4
3.1	libsark 环境搭建	4
3.2	实验代码	7
3.2.1	common.hpp	7
3.2.2	mysetup.cpp	10
3.2.3	myprove.cpp	10
3.2.4	myverify.cpp	12
3.3	运行结果	13
4	实验心得	15

图表

图 3.1.1:	安装结果	5
图 3.1.2:	安装过程	7
图 3.3.3:	实验结果	15

1 实验目的

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 $x^3 + x + 5 = \text{out}$, 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。

2 实验原理

在零知识证明中, 将待证明的命题表达为 R1CS (Rank 1 Constraint System, 一阶约束系统) 是一种常见的方法。R1CS 是一种用于表示算术关系的系统, 它可以转化为算术电路来验证计算的正确性, 这些方程构成了我们的 R1CS。

R1CS (Rank 1 Constraint System, 一阶约束系统)

R1CS (Rank 1 Constraint System, 一阶约束系统) 是一种用来表示和验证计算的数学框架, 特别是在零知识证明领域中常用来表示计算问题。R1CS 使得可以有效地证明某些计算是正确的, 而不需要揭示计算本身的细节或输入数据。R1CS 是由一组线性方程组成的, 它们描述了一个或多个多项式方程的约束。每个线性方程可以看作是对输入变量、中间变量和输出变量之间关系的描述。R1CS 的核心在于将复杂的算术表达式分解为一系列简单的线性方程。R1CS 主要包含三个部分:

变量: 这些变量包括输入变量、输出变量和中间变量。输入变量是外部提供给系统的值, 输出变量是计算结果, 而中间变量用于电路内部计算。

约束: 这些是形式为 $a * b = c$ 的方程, 其中 a 、 b 和 c 是变量或常数。每个约束描述了变量之间的一个特定关系, 通常代表电路中的一个门 (加法或乘法门)。

目标: 定义了一个或多个输出, 这些输出是满足所有约束的计算结果。

R1CS 为表示和验证计算提供了一种灵活且高效的方法, 特别适合于需要保护隐私和安全的场景。

算数电路的构建

首先，我们需要将方程转换为算术电路的形式。算术电路由输入变量、加法门、乘法门和输出组成。我们的目标是构建一个电路，该电路的输出是方程的左侧和右侧之差，即 $x^3 + x + 5 - 35$ 。

为此我们可以定义如下变量：

```
1 pb_variable<FieldT> x;
2 pb_variable<FieldT> sym_1;
3 pb_variable<FieldT> y;
4 pb_variable<FieldT> sym_2;
5 pb_variable<FieldT> out;
```

cpp

输入：变量 x 。

操作：

1. $x * x = \text{sym_1}$ 。
2. $\text{sym_1} * x = y$ 。
3. $y + x = \text{sym_2}$ 。
4. $\text{sym_2} + 5 = \text{out}$ 。

3 实验过程

3.1 libsnark 环境搭建

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，分别下载 ZIP。
- 5) 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。选择对应的 Linux 系统，执行以下命令：

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

cmd

结果如下图：

```
qmj@qmj-virtual-machine: ~/study/Libsnark/libsnark_abc-master
正在选中未选择的软件包 libgmp3-dev:amd64。
准备解压 .../6-libgmp3-dev_2%3a6.2.1+dfsg-3ubuntu1_amd64.deb ...
正在解压 libgmp3-dev:amd64 (2:6.2.1+dfsg-3ubuntu1) ...
正在选中未选择的软件包 libprocps-dev:amd64。
准备解压 .../7-libprocps-dev_2%3a3.3.17-6ubuntu2.1_amd64.deb ...
正在解压 libprocps-dev:amd64 (2:3.3.17-6ubuntu2.1) ...
正在选中未选择的软件包 python3-markdown。
准备解压 .../8-python3-markdown_3.3.6-1_all.deb ...
正在解压 python3-markdown (3.3.6-1) ...
正在设置 libboost1.74-dev:amd64 (1.74.0-14ubuntu3) ...
正在设置 libboost-program-options1.74.0:amd64 (1.74.0-14ubuntu3) ...
正在设置 libboost-program-options1.74-dev:amd64 (1.74.0-14ubuntu3) ...
正在设置 libgmpxx4ldbl:amd64 (2:6.2.1+dfsg-3ubuntu1) ...
正在设置 libboost-program-options-dev:amd64 (1.74.0.3ubuntu7) ...
正在设置 libssl-dev:amd64 (3.0.2-0ubuntu1.15) ...
正在设置 python3-markdown (3.3.6-1) ...
正在设置 libprocps-dev:amd64 (2:3.3.17-6ubuntu2.1) ...
正在设置 libgmp-dev:amd64 (2:6.2.1+dfsg-3ubuntu1) ...
正在设置 libgmp3-dev:amd64 (2:6.2.1+dfsg-3ubuntu1) ...
正在处理用于 man-db (2.10.2-1) 的触发器 ...
正在处理用于 libc-bin (2.35-0ubuntu3.5) 的触发器 ...
qmj@qmj-virtual-machine:~/study/Libsnark/libsnark_abc-master$
```

图 3.1.1: 安装结果

接下来，分别安装各个子模块，各命令如下：

编译安装各个模块

安装各个子模块方法:

安装子模块 xbyak

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

```
1 sudo make install
```

cmd

安装子模块 ate-pairing

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

```
1 make -j
2 test/bn
```

cmd

安装子模块 libsark-supercop

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

```
1 ./do
```

cmd

安装子模块 gtest

将下载得到的文件夹 Gtest 内的文件复制到 /Libsnark/libsark_abc-master/depends/libsark/depends/gtest

安装子模块 libff

将下载得到的文件夹 Libff 内的文件复制到 /Libsnark/libsark_abc-master/depends/libsark/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/libsark/depends/libff 下打开终端, 执行命令:

```
1 mkdir build
2 cd build
3 cmake ..
4 make
5 sudo make install
```

cmd

安装完之后检测是否安装成功, 执行以下命令

```
1 make check
```

cmd

安装子模块 libfqfft

将下载得到的文件夹 Libfqfft 内的文件复制到 /Libsnark/libsark_abc-master/depends/libsark/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 xbyak 也是空的, 需要将下载得到的 pairing 和 Xbyak 文件夹内的文件复制进去。在 /Libsnark/libsnark_abc-master/depends/libsark/depends/libfqfft 下打开终端, 执行命令:

```
1 mkdir build
2 cd build
3 cmake ..
4 make
5 sudo make install
```

cmd

安装完之后检测是否安装成功, 执行以下命令

```
1 make check
```

cmd

libsark 编译安装

在 /Libsnark/libsark_abc-master/depends/libsark 下打开终端, 执行以下命令:

```
1 mkdir build
2 cd build
3 cmake ..
4 make
5 make check
```

cmd

整体编译安装

在 /Libsnark/libsnark_abc-master 下打开终端, 执行以下命令:

```
1 mkdir build
2 cd build
3 cmake ..
4 make
5 ./src/test
```

cmd

安装结果如下各图展示：

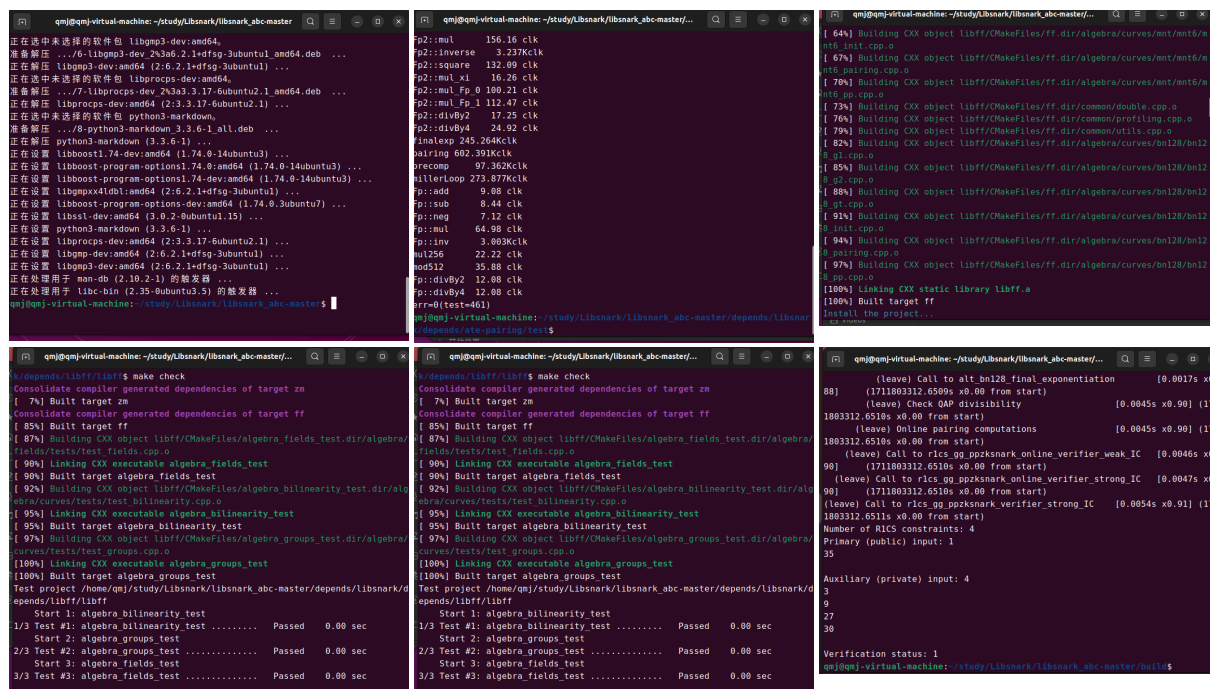
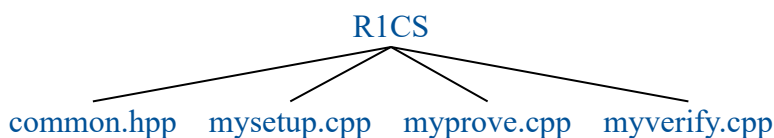


图 3.1.2: 安装过程

3.2 实验代码

我们需要完成 4 份代码：



3.2.1 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。

```

// 代码开头引用了三个头文件：第一个头文件是为了引入
1 default_rlcs_gg_ppzksnark_pp 类型；第二个则为了引入证明相关的各个接口； hpp
   pb_variable 则是用来定义电路相关的变量。

2 #include <libsark/common/default_types/rlcs_gg_ppzksnark_pp.hpp>
3 #include <libsark/zk_proof_systems/ppzksnark/rlcs_gg_ppzksnark/
   rlcs_gg_ppzksnark.hpp>
4 #include <libsark/gadgetlib1/pb_variable.hpp>
5 using namespace libsark;
6 using namespace std;
7 constexpr auto primary_input = 35;
8 // 定义使用的有限域
9 typedef libff::Fr<default_rlcs_gg_ppzksnark_pp> FieldT;
10 // 定义创建面包板的函数
  
```

```

11 protoboard<FieldT> build_protoboard(int *secret)
12 {
13     // 初始化曲线参数
14     default_rlcs_gg_ppzksnark_pp::init_public_params();
15     // 创建面包板
16     protoboard<FieldT> pb;
17     // 定义所有需要外部输入的变量以及中间变量
18     pb_variable<FieldT> x;
19     pb_variable<FieldT> sym_1;
20     pb_variable<FieldT> y;
21     pb_variable<FieldT> sym_2;
22     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。
23 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。
24     out.allocate(pb, "out");
25     x.allocate(pb, "x");
26     sym_1.allocate(pb, "sym_1");
27     y.allocate(pb, "y");
28     sym_2.allocate(pb, "sym_2");

    // 定义公有的变量的数量，set_input_sizes(n)用来声明与 protoboard 连接的 public
29 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）。
30     pb.set_input_sizes(1);
31     // 为公有变量赋值
32     pb.val(out) = primary_input;

    // 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
33 关系。
34
35     // Add R1CS constraints to protoboard
36
37     //  $x * x = \text{sym\_1}$ 
38     pb.add_rlcs_constraint(rlcs_constraint<FieldT>(x, x, sym_1));
39
40     //  $\text{sym\_1} * x = y$ 
41     pb.add_rlcs_constraint(rlcs_constraint<FieldT>(sym_1, x, y));
42
43     //  $y + x = \text{sym\_2}$ 
44     pb.add_rlcs_constraint(rlcs_constraint<FieldT>(y + x, 1, sym_2));
45
46     //  $\text{sym\_2} + 5 = \sim\text{out}$ 
47     pb.add_rlcs_constraint(rlcs_constraint<FieldT>(sym_2 + 5, 1, out));
48

```



```
49 // 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
50 if (secret != NULL)
51 {
52     pb.val(out) = secret[0];
53
54     pb.val(x) = secret[1];
55     pb.val(sym_1) = secret[2];
56     pb.val(y) = secret[3];
57     pb.val(sym_2) = secret[4];
58 }
59 return pb;
60 }
```

解析

这里定义了五个变量，分别是 `x`、`sym_1`、`y`、`sym_2` 和 `out`，它们的型是 `pb_variable`，其中 `FieldT` 是有限域类型。这些变量是用来描述一个电路的输入、输出和中间变量的。在这个示例中，`x`、`sym_1`、`y` 和 `sym_2` 是电路的中间变量，`out` 是电路的输出。这些变量的值可以在程序运行时被赋值，也可以在生成证明时被赋值。

然后我们使用 R1CS 描述电路。根据上述四个等式，我们可以得到四个约束：

```
1 // x*x = sym_1
2 pb.add_r1cs_constraint(r1cs_constraint<FieldT>(x, x, sym_1));
3 // sym_1 * x = y
4 pb.add_r1cs_constraint(r1cs_constraint<FieldT>(sym_1, x, y));
5 // y + x = sym_2
6 pb.add_r1cs_constraint(r1cs_constraint<FieldT>(y + x, 1, sym_2));
7 // sym_2 + 5 = ~out
8 pb.add_r1cs_constraint(r1cs_constraint<FieldT>(sym_2 + 5, 1, out));
```

最后，我们生成证明时为私密变量赋值。如果 `secret` 不为 `NULL`，说明当前处于生成证明的阶段，此时需要为私密变量赋值。具体地，通过 `pb.val(x) = secret[0]` 的方式为变量 `x` 赋值，`pb.val(sym_1) = secret[1]` 的方式为变量 `sym_1` 赋值，以此类推。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。

3.2.2 mysetup.cpp

```

1  #include <libsark/common/default_types/r1cs_gg_ppzksnark_pp.hpp>
2  #include <libsark/zk_proof_systems/ppzksnark/r1cs_gg_ppzksnark/
   r1cs_gg_ppzksnark.hpp>
3  #include <fstream>
4  #include "common.hpp"
5  using namespace libsark;
6  using namespace std;
7  int main()
8  {
9      // 构造面包板
10     protoboard<FieldT> pb = build_protoboard(NULL);
11     const r1cs_constraint_system<FieldT> constraint_system =
   pb.get_constraint_system();
12     // 生成证明密钥和验证密钥
13     const r1cs_gg_ppzksnark_keypair<default_r1cs_gg_ppzksnark_pp> keypair =
14         r1cs_gg_ppzksnark_generator<default_r1cs_gg_ppzksnark_pp>(constraint_
15     // 保存证明密钥到文件 pk.raw
16     fstream pk("pk.raw", ios_base::out);
17     pk << keypair.pk;
18     pk.close();
19     // 保存验证密钥到文件 vk.raw
20     fstream vk("vk.raw", ios_base::out);
21     vk << keypair.vk;
22     vk.close();
23     return 0;
24 }
25

```

解析

以上是生成公钥的初始设置阶段（Trusted Setup）。在这个阶段，我们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。这里的代码不需要改动。

3.2.3 myprove.cpp

```

1  #include <libsark/common/default_types/r1cs_gg_ppzksnark_pp.hpp>
2  #include <libsark/zk_proof_systems/ppzksnark/r1cs_gg_ppzksnark/
   r1cs_gg_ppzksnark.hpp>
3  #include <fstream>

```

```
4  #include <cmath>
5  #include "common.hpp"
6  using namespace libsark;
7  using namespace std;
8  int main()
9  {
10     // 为私密输入提供具体数值
11     double t = (5-primary_input)/2.;
12     double delta = sqrt(t*t+1/27.);
13     double res = pow(-t+delta,1/3.)-pow(t+delta,1/3.);
14     int x = round(res);
15     int secret[5];
16     secret[0] = primary_input;
17     secret[1] = x;
18     secret[2] = x*x;
19     secret[3] = x*x*x;
20     secret[4] = x*x*x+x;
21     // 构造面包板
22     protoboard<FieldT> pb = build_protoboard(secret);
23     const r1cs_constraint_system<FieldT> constraint_system =
    pb.get_constraint_system();
24     cout << "公有输入: " << pb.primary_input() << endl;
25     cout << "私密输入: " << pb.auxiliary_input() << endl;
26     // 加载证明密钥
27     fstream f_pk("pk.raw", ios_base::in);
28     r1cs_gg_ppzksnark_proving_key<libff::default_ec_pp> pk;
29     f_pk >> pk;
30     f_pk.close();
31     // 生成证明
32     const r1cs_gg_ppzksnark_proof<default_r1cs_gg_ppzksnark_pp> proof =
33         r1cs_gg_ppzksnark_prover<default_r1cs_gg_ppzksnark_pp>(
34             pk, pb.primary_input(), pb.auxiliary_input());
35     // 将生成的证明保存到 proof.raw 文件
36     fstream pr("proof.raw", ios_base::out);
37     pr << proof;
38     pr.close();
39     cout << pb.primary_input() << endl;
40     cout << pb.auxiliary_input() << endl;
41     return 0;
42 }
```

解析

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。

这里针对我们的命题, **重点编写了以下部分**:

```
1 // 为私密输入提供具体数值
2 double t = (5-primary_input)/2.;
3 double delta = sqrt(t*t+1/27.);
4 double res = pow(-t+delta,1/3.)-pow(t+delta,1/3.);
5 int x = round(res);
6 int secret[5];
7 secret[0] = primary_input;
8 secret[1] = x;
9 secret[2] = x*x;
10 secret[3] = x*x*x;
11 secret[4] = x*x*x+x;
```

这与前面在 common.hpp 之中的生成证明的部分:

```
1 pb.val(out) = secret[0];
2 pb.val(x) = secret[1];
3 pb.val(sym_1) = secret[2];
4 pb.val(y) = secret[3];
5 pb.val(sym_2) = secret[4];
```

是一一对应的。

3.2.4 myverify.cpp

```
1 #include <libsark/common/default_types/r1cs_gg_ppzksnark_pp.hpp>
2 #include <libsark/zk_proof_systems/ppzksnark/r1cs_gg_ppzksnark/
  r1cs_gg_ppzksnark.hpp>
3 #include <fstream>
4 #include "common.hpp"
5 using namespace libsark;
6 using namespace std;
7 int main()
8 {
9     // 构造面包板
10     protoboard<FieldT> pb = build_protoboard(NULL);
11     const r1cs_constraint_system<FieldT> constraint_system =
    pb.get_constraint_system();
```

```

12      // 加载验证密钥
13      fstream f_vk("vk.raw", ios_base::in);
14      rlcs_gg_ppzksnark_verification_key<libff::default_ec_pp> vk;
15      f_vk >> vk;
16      f_vk.close();
17      // 加载银行生成的证明
18      fstream f_proof("proof.raw", ios_base::in);
19      rlcs_gg_ppzksnark_proof<libff::default_ec_pp> proof;
20      f_proof >> proof;
21      f_proof.close();
22      // 进行验证
23      bool verified =
        rlcs_gg_ppzksnark_verifier_strong_IC<default_rlcs_gg_ppzksnark_pp>(vk,
          pb.primary_input(), proof);
24      cout << "验证结果:" << verified << endl;
25      return 0;
26  }

```

解析

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。编写代码如下，将这段代码放在 myverify.cpp 中。

3.3 运行结果

为了编译运行代码，我们先编写一个 CmakeLists.txt 文件，内容如下：

```

1  include_directories(.)
2  add_executable(
3      main
4      main.cpp
5  )
6  target_link_libraries(
7      main
8      snark
9  )
10 target_include_directories(
11     main
12     PUBLIC
13     ${DEPENDS_DIR}/libsna
14     ${DEPENDS_DIR}/libsna
15     depends/libfqqft

```

```

16 add_executable(
17     test
18     test.cpp
19 )
20 target_link_libraries(
21     test
22     snark
23 )
24 target_include_directories(
25     test
26     PUBLIC
27     ${DEPENDS_DIR}/libsna
28     ${DEPENDS_DIR}/libsna
29     depends/libfqqft
30 add_executable(

```

```
31 range
32 range.cpp
33 )
34 target_link_libraries(
35 range
36 snark
37 )
38 target_include_directories(
39 range
40 PUBLIC
41 ${DEPENDS_DIR}/libsnark
42     ${DEPENDS_DIR}/libsnark/
43     depends/libfqfft
44 )
45 add_executable(
46 mysetup
47 mysetup.cpp
48 )
49 target_link_libraries(
50 mysetup
51 snark
52 )
53 target_include_directories(
54 mysetup
55 PUBLIC
56 ${DEPENDS_DIR}/libsnark
57     ${DEPENDS_DIR}/libsnark/
58     depends/libfqfft
59 )
60 add_executable(
61 myprove
62 myprove.cpp
63 )
64 target_link_libraries(
65 myprove
66 snark
67 )
68 target_include_directories(
69 myprove
70 PUBLIC
71 ${DEPENDS_DIR}/libsnark
72     ${DEPENDS_DIR}/libsnark/
73     depends/libfqfft
```

```
71 )
72 add_executable(
73 myverify
74 myverify.cpp
75 )
76 target_link_libraries(
77 myverify
78 snark
79 )
80 target_include_directories(
81 myverify
82 PUBLIC
83 ${DEPENDS_DIR}/libsnark
84     ${DEPENDS_DIR}/libsnark/
85     depends/libfqfft
86 )
```

CmakeLists.txt 文件的主要功能是配置项目结构。首先,它使用 `include_directories` 添加当前目录到头文件搜索路径。接着,通过 `add_executable` 增加了五个可执行文件: `main`、`test`、`range`、`mysetup` 和 `myprove`。每个文件都通过 `target_link_libraries` 与 `snark` 库关联。为了确保库文件的正确包含, `target_include_directories` 被用来添加 `libsark` 和 `libfqfft` 库的路径。最后,对每个可执行文件,都按照相同的模式设置了链接和包含路径,以确保编译时的兼容性。

接下来依次运行命令:

```
1 cmake ..
2 make
3 cd src
4 ./mysetup
5 ./myprove
6 2
7 ./myverify
```

实验结果

结果如下图,证明我们的代码正确,验证成功:

```
qmj@qmj-virtual-machine: ~/study/Libsnark/libsnark_abc-master/...
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
qmj@qmj-virtual-machine: ~/study/Libsnark/libsnark_abc-master/build$

qmj@qmj-virtual-machine: ~/study/Libsnark/libsnark_abc-master/build/src$
(enter) Call to alt_bn128_exp_by_neg_z [
(1711806597.3517s x0.00 from start)
(leave) Call to alt_bn128_exp_by_neg_z [0.0004s x1.00]
(1711806597.3521s x0.00 from start)
(enter) Call to alt_bn128_exp_by_neg_z [
(1711806597.3522s x0.00 from start)
(leave) Call to alt_bn128_exp_by_neg_z [0.0004s x1.00]
(1711806597.3526s x0.00 from start)
(leave) Call to alt_bn128_final_exponentiation_last_chunk [0.0014s x1.00]
(1711806597.3527s x0.00 from start)
(leave) Call to alt_bn128_final_exponentiation [0.0015s x1.00]
(1711806597.3527s x0.00 from start)
(leave) Check QAP divisibility [0.0041s x1.00] (1711806597.3527s x0.00 from start)
(leave) Online pairing computations [0.0041s x1.00] (1711806597.3527s x0.00 from start)
(leave) Call to r1cs_gg_ppzksnark_online_verifier_weak_IC [0.0041s x1.00]
(1711806597.3527s x0.00 from start)
(leave) Call to r1cs_gg_ppzksnark_online_verifier_strong_IC [0.0041s x1.00]
(1711806597.3527s x0.00 from start)
(leave) Call to r1cs_gg_ppzksnark_verifier_strong_IC [0.0049s x1.00] (1711806597.3527s x0.00 from start)
验证结果:1
```

图 3.3.3: 实验结果

验证结果为 1, 表示通过验证

4 实验心得

通过这次实验,我深入了解了零知识证明和 R1CS 的核心概念及其应用。实现方程 $x^3 + x + 5 = 35$ 的零知识证明过程,让我实际体验了如何将复杂的多项式方程转换为 R1CS,并通过构建算术电路来形成一套约束系统。这个过程不仅加深了我对算术电路和零知识证明技术的理解,也让我认识到这些技术在保护隐私和数据安全方面的巨大潜

力。此外，实践中遇到的挑战和问题解决经历，增强了我的问题分析和解决能力，对我的学术和职业生涯都是宝贵的财富。