南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

数据安全

零知识证明实践

穆禹宸　　2012026

年级：2020 级

专业：信息安全、法学双学位班

指导教师：刘哲理

2023 年 4 月 11 日

目录

一、 实验名称

零知识证明实践

二、 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：$x^3+x+5=out$。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。

三、 实验过程

实验代码

本次实验所需要编写的代码一共有五个文件：

(1) common.hpp

(2) mysetup.cpp

(3) myprove.cpp

(4) myverify.cpp

(5) CMakeLists.txt

其中，`common.hpp` 和 `myprove.cpp` 是本次实验需要重点编写的代码！

(一) 实验代码

1. common.hpp

这个代码的核心内容，是在用 R1CS 描述电路之后，就构建一个 protoboard。由于在初始设置、证明、验证三个阶段都需要构造面包板，因此本段代码写成共有头文件的形式。

完整代码如下所示：

common.hpp

```
// 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> sym_1;
    pb_variable<FieldT> y;
    pb_variable<FieldT> sym_2;
    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");
    sym_1.allocate(pb, "sym_1");
    y.allocate(pb, "y");
    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 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) = 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。

    // Add R1CS constraints to protoboard

    // x*x = sym_1
    pb.add_r1cs_constraint(r1cs_constraint<FieldT>(x, x, sym_1));

    // sym_1 * x = y
    pb.add_r1cs_constraint(r1cs_constraint<FieldT>(sym_1, x, y));

    // y + x = sym_2
    pb.add_r1cs_constraint(r1cs_constraint<FieldT>(y + x, 1, sym_2));

    // sym_2 + 5 = ~out
    pb.add_r1cs_constraint(r1cs_constraint<FieldT>(sym_2 + 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];
```

```
55                  pb.val(sym_1) = secret[1];
56                  pb.val(y) = secret[2];
57                  pb.val(sym_2) = secret[3];
58      }
59      return pb;
60  }
```

其中我们重点关注如下部分:

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

这里定义了五个变量，分别是 x、sym_1、y、sym_2 和 out，它们的类型是 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。

一共有如下等式:

R1CS

一共有四个等式:

(1) $x * x = sym_1$

(2) $sym_1 * x = y$

(3) $y + x = sym_2$

(4) $sym_2 + 5 = out$

至此，整个电路被拍平在面包板上。

然后，针对四个等式，写出如下代码:

```
// Add R1CS constraints to protoboard

// x*x = sym_1
pb.add_r1cs_constraint(r1cs_constraint<FieldT>(x, x, sym_1));

// sym_1 * x = y
pb.add_r1cs_constraint(r1cs_constraint<FieldT>(sym_1, x, y));

// y + x = sym_2
pb.add_r1cs_constraint(r1cs_constraint<FieldT>(y + x, 1, sym_2));

// sym_2 + 5 = ~out
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。

```
// 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(sym_1) = secret[1];
    pb.val(y) = secret[2];
    pb.val(sym_2) = secret[3];
}
return pb;
```

至此，针对命题的电路已构建完毕。

2. mysetup.cpp

接下来，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。

mysetup.cpp

```
#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;
}
```

也就是说，本段程序通过 r1cs_gg_ppzksnark_generator 函数生成了证明密钥和验证密钥，并将它们分别保存到 pk.raw 和 vk.raw 文件中。

3. myprove.cpp

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。完整代码如下所示:

myprove.cpp

```
#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>
#include "common.hpp"
using namespace libsnark;
using namespace std;
int main()
{
    // 为私密输入提供具体数值
    int x;
    cout<<"请输入x:";
    cin>>x;
    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();
    cout << pb.primary_input() << endl;
    cout << pb.auxiliary_input() << endl;
    return 0;
}
```

这里针对我们的命题，重点编写了以下部分：

```
// 为私密输入提供具体数值
int x;
cout<<" 请输入 x:";
cin>>x;
int secret[4];
secret[0] = x;
secret[1] = x*x;
secret[2] = x*x*x;
secret[3] = x*x*x+x;
```

注意，回顾之前在 common.hpp 之中的生成证明的部分：

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

注意是一一对应的关系。

4. myverify.cpp

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。本段代码也不需要改动。

myverify.cpp

```cpp
#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;
}
```

至此，全部代码编写结束。

(二) 实验结果展示

编写 CMakeLists.txt，内容如下所示:

CMakeLists.txt

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

这段代码的大致解释是：通过 include_directories 函数将当前目录加入到 include 路径中，通过 add_executable 函数添加了五个可执行文件 main、test、range、mysetup 和 myprove，并通过 target_link_libraries 函数将这些可执行文件与 snark 库链接。接着，通过 target_include_directories 函数将 libsnark 和 libfqfft 库的路径添加到 include 路径中。最后，每个可执行文件都通过类似的方式进行了链接和 include 路径的设置。

然后运行如下命令：

```
cmake ..
make
cd src
./mysetup
./myprove
3
./myverify
```

在输入 x=3 之后，出现了如下数字。

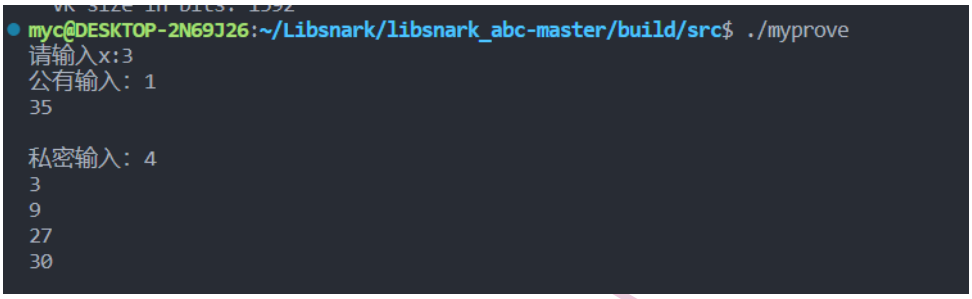

图 1: 输入私密 x=3

这里我们注意，公有输入: 1 和 私密输入:4 的含义如下所示：

pb.primary_input() 代表了零知识证明中的主要输入，也就是需要证明的语句的输入。在零知识证明中，我们需要证明某个语句是正确的，但是不需要暴露语句的具体内容，只需要证明我们知道语句的正确性即可。而 pb.primary_input() 就是这个需要证明的语句的输入。也就是说，有一个语句，这一个公有输入即为 35 ，即 out 为 35 需要被验证。

而 pb.auxiliary_input() 是零知识证明中的辅助输入，通常是私有的输入。在零知识证明中，证明者需要证明他知道满足某个语句的私有输入，同时不泄露这个私有输入的具体内容。也就是说我们有四个输入（**四个约束**），而后分别是四个（包括中间变量）私有输入的内容了。

最后，我们得到如下结果：

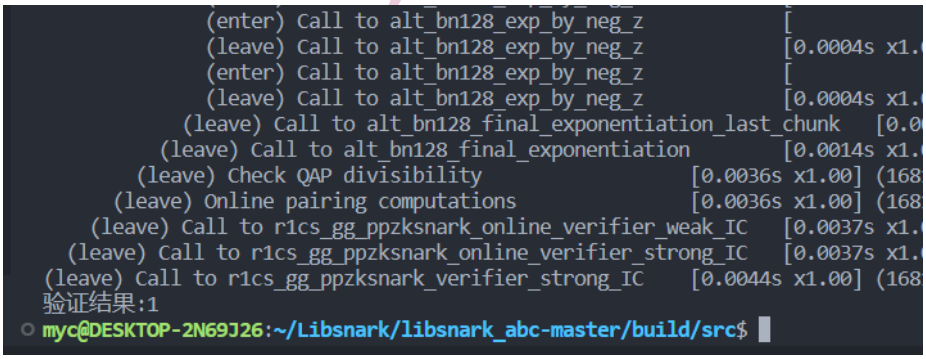

图 2: 最终结果

可以看到，输出的结果为 1 ！也就是说通过验证！至此，实验圆满成功！

过程之中其他输出。

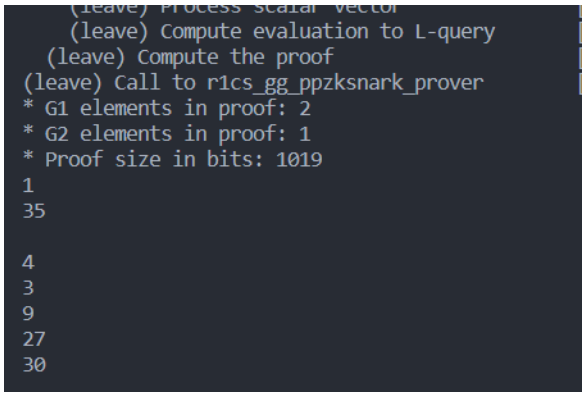

图 3: 过程之中展示的 log

四、 心得体会

这个零知识证明实现是一个基于 R1CS（Rank-1 Constraint System）的案例，使用了 libsnark 库来实现。在这个实现中，我们使用了 r1cs_gg_ppzksnark 算法来生成零知识证明。在实验中，我学习了如何使用 libsnark 库来实现零知识证明，并对 R1CS 约束系统有了更深入的了解。

我需要了解 R1CS 约束系统的基本概念。R1CS 约束系统是一种用于描述约束关系的数据结构，它由一组约束关系和一组公有变量组成。

其次，我需要了解如何使用 libsnark 库来实现零知识证明。在这个例子中，我们使用了 r1cs_gg_ppzksnark 算法来生成零知识证明。这个算法基于双线性对，可以用于生成零知识证明，并且具有高效性和安全性。在实验中，我学习了如何加载证明密钥、生成证明和验证证明的正确性。这些操作都是通过调用 libsnark 库中的函数来实现的。

最后，我需要了解如何使用面包板（protoboard）来描述约束系统。面包板是一个用于描述约束系统的数据结构，它由一组变量和一组约束关系组成。在这个例子中，我们使用 protoboard 类型的对象来描述约束系统。我们首先定义了需要外部输入的变量和中间变量，然后将这些变量与 protoboard 对象连接起来。接着，我们定义了变量之间的约束关系，并将这些约束关系添加到 protoboard 对象中。最后，我们根据传入的私密输入为私密变量赋值。

在实验中，我遇到了一些困难，比如理解 R1CS 约束系统的概念、学习如何使用 libsnark 库以及调试代码等。但是通过不断地学习和尝试，我最终成功地生成了零知识证明，并验证了证明的正确性。在这个过程中，我学习到了很多关于零知识证明和 libsnark 库的知识，也提高了自己的编程能力和解决问题的能力。

当然，libsnark 库的安装也废了很大功夫。

总之，这个实验让我更深入地了解了零知识证明和 libsnark 库，并提高了我的编程能力和解决问题的能力。我相信这些知识和能力在未来的学习和工作中都会对我有很大的帮助。