



南開大學
Nankai University

计算机学院
数据安全实验报告

零知识证明实践

姓名：李欣

学号：2011165

专业：计算机科学与技术

2023 年 4 月 3 日

目录

1 实验目的	2
2 开发环境	2
3 零知识证明	2
3.1 基本概念	2
3.2 交互式 Schnorr 协议	3
3.3 非交互式零知识证明	3
3.3.1 设计要点	3
3.3.2 非交互式 Schnorr 协议	3
3.3.3 简洁零知识证明 zkSNARK	4
4 libsnark 框架	4
4.1 框架概述	4
4.2 环境搭建	5
5 代码实现与实验结果	10
5.1 将待证明的命题表达为 R1CS	10
5.2 生成证明密钥和验证密钥	12
5.3 证明方使用证明密钥和其可行解构造证明	13
5.4 验证方使用验证密钥验证证明方发过来的证明	14
5.5 结果展示	15
6 总结	15

1 实验目的

学习零知识证明相关理论，包括交互式 Schnorr 协议，非交互式 Schnorr 协议和简洁零知识证明 zkSNARK。

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 是大家都已知的一个数，这里假设 out 为 35，而 $x=3$ 就是方程的解，借助 libsnark 框架实现代码完成证明生成和证明的验证。

2 开发环境

- VMware Workstation 16
- Ubuntu20.04/64

3 零知识证明

首先我们对零知识证明的理论知识进行学习，主要包含交互式零知识证明和非交互式零知识证明，在最后，我们重点学习了实现实验需要的理论知识简介零知识证明 zkSNARK。

3.1 基本概念

零知识证明是由 S.Goldwasser、S.Micali 及 C.Rackoff 在 20 世纪 80 年代初提出的，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。

1. 定义

零知识 (Zero-Knowledge, ZK) 证明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。

这里的 C 是一个公开的谓词函数。谓词函数是一个判断式，一个返回 bool 值的函数。该定义意味着一个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。

2. 性质

(a) 正确性

没有人能够假冒证明方 P 使这个证明成功。如果不满足这条性质，也就是证明方 P 不知道“知识”，再怎么证明，验证方 V 也很难相信证明方 P 拥有正确的知识。

(b) 完备性

如果证明方 P 和验证方 V 都是诚实的，并证明过程的每一步都进行正确的计算，那么这个证明一定是成功的。也就是说如果证明方 P 知道“知识”，那么验证方 V 会有极大的概率相信证明方 P 。

(c) 零知识性

证明执行完之后，验证方 V 只获得了“证明方 P 拥有这个知识”这条信息，而没有获得关于这个知识本身的任何信息。

3.2 交互式 Schnorr 协议

交互式协议不是本次实验的重点，在这里，笔者仅作参考。

交互式零知识证明需要证明者和验证者时刻保持在线状态，而这会因网络延迟、拒绝服务等原因难以保障。

交互式零知识证明的协议流程分为以下四个步骤：

1. 承诺阶段
2. 挑战阶段
3. 回应阶段
4. 验证阶段

而交互式协议的安全性由椭圆曲线上的离散对数问题保证。

3.3 非交互式零知识证明

3.3.1 设计要点

在非交互式零知识证明 (Non-Interactive Zero Knowledge, NIZK) 中，证明者仅需发送一轮消息即可完成证明

为保证协议的安全性，即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。

3.3.2 非交互式 Schnorr 协议

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 变换可以将交互式零知识证明转换为非交互式零知识证明，实现思路是用公开的哈希函数的输出代替随机的挑战。

- 协议流程

1. 承诺阶段

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 $z*G \stackrel{?}{=} R+c*PK$ 。

- 安全性

通过使用随机预言机（种针对任意输入得到的输出是独立且均匀分布的哈希函数），使得 Alice 只能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，保证安全性。

3.3.3 简洁零知识证明 zkSNARK

基于公共参考字符串 CRS 实现非交互式零知识证明技术是本次实验的理论基础。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。

1. 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。
- 零知识：验证者无法获取证明者的任何隐私信息。

2. 开发步骤

- 定义电路：将所要声明的内容的计算算法用算术电路来表示，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。所有的 NP 问题都可以有效地转换为算术电路。
- 将电路表达为 R1CS：在电路的基础上构造约束，也就是 R1CS (Rank-Constraint System, 一阶约束系统)，有了约束就可以把 NP 问题抽象成 QAP (Quadratic Arithmetic Problem) 问题。R1CS 与 QAP 形式上的区别是 QAP 使用多项式来代替点积运算，而它们的实现逻辑完全相同。有了 QAP 问题的描述，就可以构建 zkSNARKs。
- 完成应用开发：
 - 生成密钥：生成证明密钥 (Proving Key) 和验证密钥 (Verification Key)；
 - 生成证明：证明方使用证明密钥和其可行解构造证明；
 - 验证证明：验证方使用验证密钥验证证明方发过来的证明。

基于 zkSNARK 的实际应用，最终实现的效果就是证明者给验证者一段简短的证明，验证者可以自行校验某命题是否成立。

4 libsnark 框架

4.1 框架概述

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。

- **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 `libsark/libsark/zk_proof_systems` 路径, 就能发现 `libsark` 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/ppzksark/r1cs_ppzksark` 对应的是 BCTV14a
- `zk_proof_systems/ppzksark/r1cs_gg_ppzksark` 对应的是 Groth16

在 Groth16 中, `ppzksark` 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。

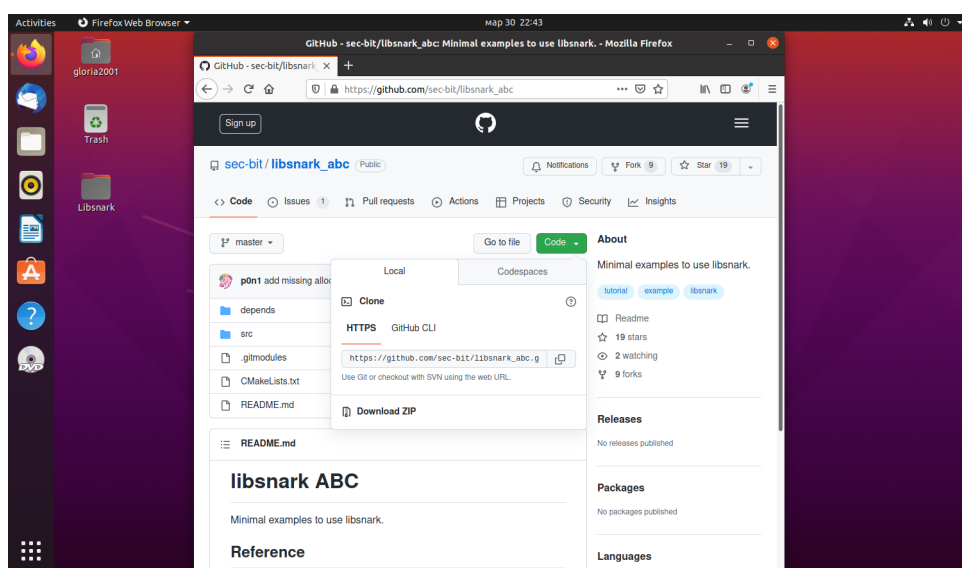
4.2 环境搭建

本次实验, 我在虚拟机 `vmware16` 上安装 `ubuntu20.04 (64)`, 作为实验的环境, 虚拟机的安装这里不做赘述, 参照课件和 `github` 官方安装文档, 安装 `libsark` 框架, 具体步骤如下:

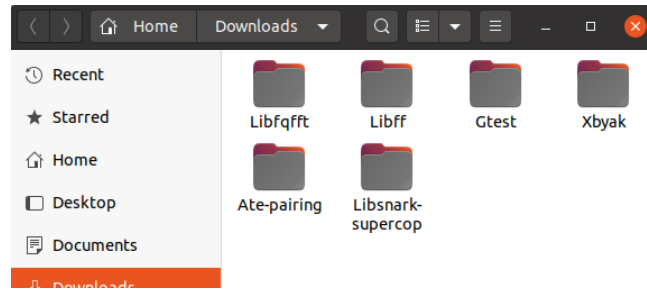
1. 安装依赖的工具

```
gloria2001@gloria2001:~/Desktop$ sudo apt install build-essential cmake git libgmp3-dev libprocps-dev python3-markdown libboost-program-options-dev libssl-dev
python3 pkg-config
[sudo] password for gloria2001:
Reading package lists... Done
Building dependency tree
Reading state information... Done
libboost-program-options-dev is already the newest version (1.71.0-0ubuntu2).
pkg-config is already the newest version (0.29.1-0ubuntu4).
python3 is already the newest version (3.8.2-0ubuntu2).
python3-markdown is already the newest version (3.1.1-3).
build-essential is already the newest version (12.8ubuntu1.1).
cmake is already the newest version (3.16.3-1ubuntu1.20.04.1).
git is already the newest version (1:2.25.1-1ubuntu3.10).
libgmp3-dev is already the newest version (2:6.2.0+dfsg-4ubuntu0.1).
libprocps-dev is already the newest version (2:3.3.16-1ubuntu2.3).
libssl-dev is already the newest version (1.1.1f-1ubuntu2.17).
0 upgraded, 0 newly installed, 0 to remove and 620 not upgraded.
```

2. 下载相应的工具包



3. 下载子模块



4. 安装子模块 xbyak

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

```
1 sudo make install
```

```
gloria2001@gloria2001:~/Desktop/Libsnark/libsnark_abc-master/depends/libsnark/depends/xbyak$ sudo make install
[sudo] password for gloria2001:
mkdir -p /usr/local/include/xbyak
cp -pR xbyak/*.h /usr/local/include/xbyak
```

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

```
1 make -j
```

```
g++ -o java_api java_api.o -lm -lzm -L../lib -lgmp -lgmpxx -m64
g++ -o loop_test loop_test.o -lm -lzm -L../lib -lgmp -lgmpxx -m64
g++ -o sample sample.o -lm -lzm -L../lib -lgmp -lgmpxx -m64
g++ -o bench_test bench.o -lm -lzm -L../lib -lgmp -lgmpxx -m64
g++ -o test_zm test_zm.o -lm -lzm -L../lib -lgmp -lgmpxx -m64
g++ -o bn bn.o -lm -lzm -L../lib -lgmp -lgmpxx -m64
gloria2001@gloria2001:~/Desktop/Libsnark/libsnark_abc-master/depends/libsnark/depends/ate-pairing$
```

```
1 test/bn
```

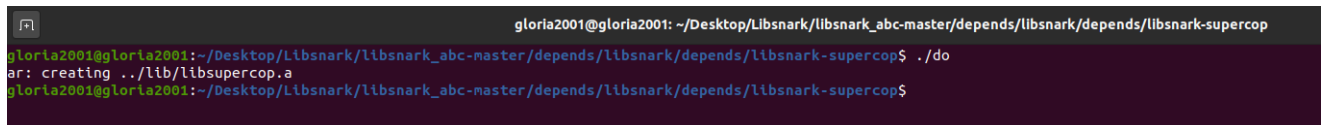
```

Fp2::divBy4    75.36 clk
finalexp 424.686Kclk
pairing 753.388Kclk
precomp    133.638Kclk
millerLoop 244.934Kclk
Fp::add      10.16 clk
Fp::sub      11.56 clk
Fp::neg       7.07 clk
Fp::mul     102.01 clk
Fp::inv     14.500Kclk
mul256       32.92 clk
mod512       59.78 clk
Fp::divBy2   46.32 clk
Fp::divBy4   34.09 clk
err=0(test=461)
gloria2001@gloria2001:~/Desktop/Libsnark/libsnark_abc-master/depends/libsnark/depends/ate-pairing$
```

6. 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 /Desktop/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

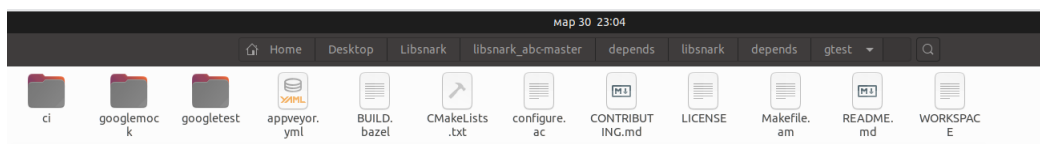
```
1  ./do
```



```
gloria2001@gloria2001: ~/Desktop/Libsnark/libsnark_abc-master/depends/libsnark/depends/libsnark-supercop
gloria2001@gloria2001:~/Desktop/Libsnark/libsnark_abc-master/depends/libsnark/depends/libsnark-supercop$ ./do
ar: creating ../lib/libsupercop.a
gloria2001@gloria2001:~/Desktop/Libsnark/libsnark_abc-master/depends/libsnark/depends/libsnark-supercop$
```

7. 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



8. 安装子模块 libff

将下载得到的文件夹 Libff 内的文件复制到 /Desktop/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 /Desktop/Libsnark/libsnark_abc-master/depends/libsnark/depends/libff 下打开终端，执行如下命令进行安装和检查：

```
1  mkdir build
2  cd build
3  cmake ..
4  make
5  sudo make install
6
7  make check//检查
```

```

gloria2001@gloria2001:~/Desktop/Libsnark/libsnark_abc-master/depends/libsnark/depends/libff/build$ make check
[ 7%] Built target zm
[ 85%] Built target ff
Scanning dependencies of target algebra_bilinearity_test
[ 87%] Building CXX object libff/CMakeFiles/algebra_bilinearity_test.dir/algebra/curves/tests/test_bilinearity.cpp.o
[ 90%] Linking CXX executable algebra_bilinearity_test
[ 90%] Built target algebra_bilinearity_test
Scanning dependencies of target algebra_fields_test
[ 92%] Building CXX object libff/CMakeFiles/algebra_fields_test.dir/algebra/fields/tests/test_fields.cpp.o
[ 95%] Linking CXX executable algebra_fields_test
[ 95%] Built target algebra_fields_test
Scanning dependencies of target algebra_groups_test
[ 97%] Building CXX object libff/CMakeFiles/algebra_groups_test.dir/algebra/curves/tests/test_groups.cpp.o
[100%] Linking CXX executable algebra_groups_test
[100%] Built target algebra_groups_test
Scanning dependencies of target check
Test project /home/gloria2001/Desktop/Libsnark/libsnark_abc-master/depends/libsnark/depends/libff/build
Start 1: algebra_bilinearity_test
1/3 Test #1: algebra_bilinearity_test ..... Passed    0.00 sec
Start 2: algebra_groups_test
2/3 Test #2: algebra_groups_test ..... Passed    0.01 sec
Start 3: algebra_fields_test
3/3 Test #3: algebra_fields_test ..... Passed    0.00 sec

100% tests passed, 0 tests failed out of 3

Total Test time (real) = 0.01 sec

```

9. 安装子模块 libfqfft

将下载得到的文件夹 Libfqfft 内的文件复制到 /Desktop/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 xbyak 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 /Desktop/Libsnark/libsnark_abc-master/depends/libsnark/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
6
7  make check//检查

```

```

[ 95%] Building CXX object libfqfft/CMakeFiles/polynomial_arithmetic_test.dir/te
sts/init_test.cpp.o
[ 97%] Building CXX object libfqfft/CMakeFiles/polynomial_arithmetic_test.dir/te
sts/polynomial_arithmetic_test.cpp.o
[100%] Linking CXX executable polynomial_arithmetic_test
[100%] Built target polynomial_arithmetic_test
Scanning dependencies of target check
Test project /home/gloria2001/Desktop/Libsnark/libsnark_abc-master/depends/libsn
ark/depends/libfqfft/build
Start 1: evaluation_domain_test
1/3 Test #1: evaluation_domain_test ..... Passed    0.00 sec
Start 2: polynomial_arithmetic_test
2/3 Test #2: polynomial_arithmetic_test ..... Passed    0.00 sec
Start 3: kronecker_substitution_test
3/3 Test #3: kronecker_substitution_test ..... Passed    0.00 sec

100% tests passed, 0 tests failed out of 3

Total Test time (real) = 0.01 sec
[100%] Built target check
gloria2001@gloria2001:~/Desktop/Libsnark/libsnark_abc-master/depends/libsnark/de
pends/libfqfft/build$

```

10. libsnark 编译安装在 /Desktop/Libsnark/libsnark_abc-master/depends/libsnark 下打开终端, 执行以下命令:

```

1  mkdir build
2  cd build
3  cmake ..
4  make
5
6  make check//检查

```

```

Start 13: zk_proof_systems_bacs_ppzksnark_test
13/23 Test #13: zk_proof_systems_bacs_ppzksnark_test ..... Passed    0.00 sec
Start 14: zk_proof_systems_rics_ppzksnark_test
14/23 Test #14: zk_proof_systems_rics_ppzksnark_test ..... Passed    0.00 sec
Start 15: zk_proof_systems_rics_se_ppzksnark_test
15/23 Test #15: zk_proof_systems_rics_se_ppzksnark_test ..... Passed    0.00 sec
Start 16: zk_proof_systems_rics_gg_ppzksnark_test
16/23 Test #16: zk_proof_systems_rics_gg_ppzksnark_test ..... Passed    0.00 sec
Start 17: zk_proof_systems_ram_ppzksnark_test
17/23 Test #17: zk_proof_systems_ram_ppzksnark_test ..... Passed    0.00 sec
Start 18: zk_proof_systems_tbcg_ppzksnark_test
18/23 Test #18: zk_proof_systems_tbcg_ppzksnark_test ..... Passed    0.01 sec
Start 19: zk_proof_systems_uscg_ppzksnark_test
19/23 Test #19: zk_proof_systems_uscg_ppzksnark_test ..... Passed    0.00 sec
Start 20: test_knapsack_gadget
20/23 Test #20: test_knapsack_gadget ..... Passed    0.00 sec
Start 21: test_merkle_tree_gadgets
21/23 Test #21: test_merkle_tree_gadgets ..... Passed    0.00 sec
Start 22: test_set_commitment_gadget
22/23 Test #22: test_set_commitment_gadget ..... Passed    9.74 sec
Start 23: test_sha256_gadget
23/23 Test #23: test_sha256_gadget ..... Passed    0.06 sec

100% tests passed, 0 tests failed out of 23

Total Test time (real) = 60.48 sec
gloria2001@gloria2001:~/Desktop/Libsnark/libsnark_abc-master/depends/libsnark/build$

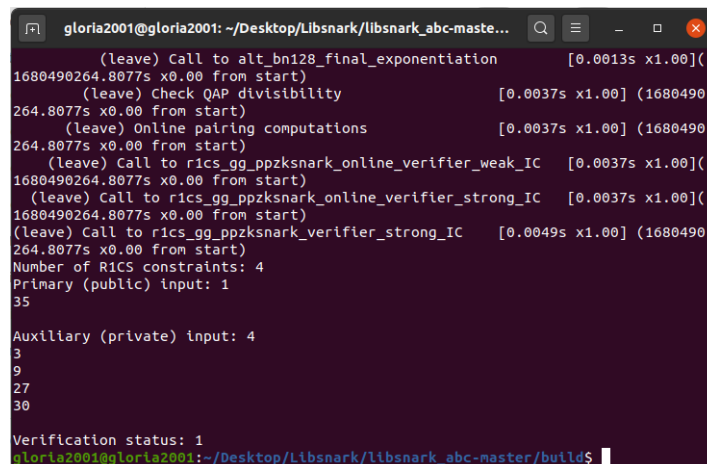
```

11. 整体编译安装在 /Desktop/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

```



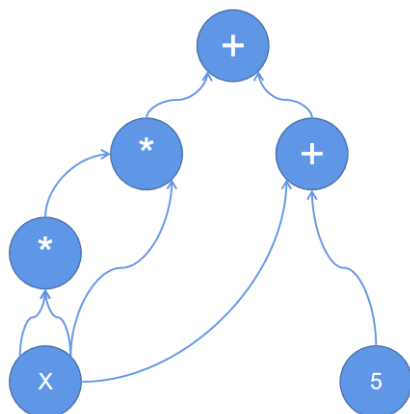
```

gloria2001@gloria2001: ~/Desktop/Libsnark/libsnark_abc-maste...
(leave) Call to alt_bn128_final_exponentiation [0.0013s x1.00](
1680490264.8077s x0.00 from start)
(leave) Check QAP divisibility [0.0037s x1.00] (1680490
264.8077s x0.00 from start)
(leave) Online pairing computations [0.0037s x1.00] (1680490
264.8077s x0.00 from start)
(leave) Call to rics_gg_ppzksnark_online_verifier_weak_IC [0.0037s x1.00](
1680490264.8077s x0.00 from start)
(leave) Call to rics_gg_ppzksnark_online_verifier_strong_IC [0.0037s x1.00](
1680490264.8077s x0.00 from start)
(leave) Call to rics_gg_ppzksnark_verifier_strong_IC [0.0049s x1.00] (1680490
264.8077s x0.00 from start)
Number of RICS constraints: 4
Primary (public) input: 1
35
Auxiliary (private) input: 4
3
9
27
30
Verification status: 1
gloria2001@gloria2001:~/Desktop/Libsnark/libsnark_abc-master/build$

```

5 代码实现与实验结果

5.1 将待证明的命题表达为 R1CS



接着，我们使用 R1CS 描述电路

$$w_1 = x * x$$

$$w_2 = x * x * x$$

$$w_3 = x + 5$$

$$out = w_2 + w_3$$

接着，使用原型板 protoboard 搭建电路

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。

```

1 [ title=common.hpp,frame=trbl ,language={C++}]
2 //代码开头引用了三个头文件:第一个头文件是为了引入 default_rlcs_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。
3 #include <libsark/common/default_types/rlcs_gg_ppzksnark_pp.hpp>
4 #include
   <libsark/zk_proof_systems/ppzksnark/rlcs_gg_ppzksnark/rlcs_gg_ppzksnark.hpp>
5 #include <libsark/gadgetlib1/pb_variable.hpp>
6 using namespace libsark;
7 using namespace std;
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> w_1;
20 pb_variable<FieldT> w_2;
21 pb_variable<FieldT> w_3;
22 // pb_variable<FieldT> w_4;
23 // pb_variable<FieldT> w_5;
24 pb_variable<FieldT> out;
25 //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。
26 out.allocate(pb, "out");
27 x.allocate(pb, "x");
28 w_1.allocate(pb, "w_1");
29 w_2.allocate(pb, "w_2");
30 w_3.allocate(pb, "w_3");
31 // w_4.allocate(pb, "w_4");
32 // w_5.allocate(pb, "w_5");
33 //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)。
34 pb.set_input_sizes(1);
35 //为公有变量赋值
36 // pb.val(out)=0;
37 //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_rlcs_constraint()函数,为 pb 添加形如  $a * b = c$  的 rlcs_constraint。即
    rlcs_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。
38 //  $x-1 = w_1$ 
39 // pb.add_rlcs_constraint(rlcs_constraint<FieldT>(x-1, 1, w_1));
40 // //  $x-2 = w_2$ 
41 // pb.add_rlcs_constraint(rlcs_constraint<FieldT>(x-2, 1, w_2));
42 // //  $x-3 = w_3$ 
43 // pb.add_rlcs_constraint(rlcs_constraint<FieldT>(x-3, 1, w_3));
44 // //  $x * w_1 = w_4$ 
45 // pb.add_rlcs_constraint(rlcs_constraint<FieldT>(x, w_1, w_4));
46 // //  $w_2 * w_4 = w_5$ 
47 // pb.add_rlcs_constraint(rlcs_constraint<FieldT>(w_2, w_4, w_5));
48 // //  $w_3 * w_5 = out$ 
49 // pb.add_rlcs_constraint(rlcs_constraint<FieldT>(w_3, w_5, out));
50 pb.val(out)=35;
51 pb.add_rlcs_constraint(rlcs_constraint<FieldT>(x, x, w_1));
52 pb.add_rlcs_constraint(rlcs_constraint<FieldT>(w_1, x, w_2));
53 pb.add_rlcs_constraint(rlcs_constraint<FieldT>(x+5, 1, w_3));
54 pb.add_rlcs_constraint(rlcs_constraint<FieldT>(w_2+w_3, 1, out));
55 // pb.add_rlcs_constraint(rlcs_constraint<FieldT>(w_4, 1, out));

```

```

56 //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
57 if (secret!=NULL)
58 {
59 pb.val(x)=secret[0];
60 pb.val(w_1)=secret[1];
61 pb.val(w_2)=secret[2];
62 pb.val(w_3)=secret[3];
63 // pb.val(w_4)=secret[4];
64 // pb.val(w_5)=secret[5];
65 }
66 return pb;
67 }

```

5.2 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。

mysetup.cpp

```

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

```

23 }

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。编写代码如下，将这段代码放在 myprove.cpp 中。

myprove.cpp

```

1  #include <libsark/common/default_types/rlcs_gg_ppzksnark_pp.hpp>
2  #include
   <libsark/zk_proof_systems/ppzksnark/rlcs_gg_ppzksnark/rlcs_gg_ppzksnark.hpp>
3  #include <fstream>
4  #include "common.hpp"
5  using namespace libsark;
6  using namespace std;
7  int main()
8  {
9  //输入秘密值 x
10 int x;
11 cin>>x;
12 //为私密输入提供具体数值
13 // int secret[6];
14 // secret[0]=x;
15 // secret[1]=x-1;
16 // secret[2]=x-2;
17 // secret[3]=x-3;
18 // secret[4]=x*(x-1);
19 // secret[5]=x*(x-1)*(x-2);
20 // int secret[6];
21 int secret[5];
22 secret[0]=x;
23 secret[1]=x*x;
24 secret[2]=x*x*x;
25 secret[3]=x+5;
26 // secret[4]=x*x*x+x+5;
27 // secret[5]=x*(x-1)*(x-2);
28 //构造面包板
29 protoboard<FieldT> pb=build_protoboard(secret);
30 const rlcs_constraint_system<FieldT> constraint_system =
   pb.get_constraint_system();
31 cout<<"公有输入:"<<pb.primary_input()<<endl;
32 cout<<"私密输入:"<<pb.auxiliary_input()<<endl;
33 //加载证明密钥
34 fstream f_pk("pk.raw",ios_base::in);
35 rlcs_gg_ppzksnark_proving_key<libff::default_ec_pp>pk;
36 f_pk>>pk;
37 f_pk.close();

```

```

38 //生成证明
39 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());
40 //将生成的证明保存到 proof.raw 文件
41 fstream pr("proof.raw",ios_base::out);
42 pr<<proof;
43 pr.close();
44 return 0;
45 }

```

5.4 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。

myverify.cpp

```

1  #include <libsnark/common/default_types/r1cs_gg_ppzksnark_pp.hpp>
2  #include
    <libsnark/zk_proof_systems/ppzksnark/r1cs_gg_ppzksnark/r1cs_gg_ppzksnark.hpp>
3  #include <fstream>
4  #include "common.hpp"
5  using namespace libsnark;
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  r1cs_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  r1cs_gg_ppzksnark_proof<libff::default_ec_pp>proof;f_proof>>proof;
20  f_proof.close();
21  //进行验证
22  bool
    verified=r1cs_gg_ppzksnark_verifier_strong_IC<default_r1cs_gg_ppzksnark_pp>(vk,pb.primary_
    proof);
23  cout<<"验证结果:"<<verified<<endl;
24  return 0;
25  }

```

5.5 结果展示

输入 $x=3$

```
gloria2001@gloria2001:~/Desktop/Libsnark/libsnark_abc-master/build/src$ ./myprove
3
公有输入:1
35

私密输入:4
3
9
27
8
```

验证结果为, 表示 $x=3$ 是方程 $x^3+x+5=35$ 的解。

```
gloria2001@gloria2001:~/Desktop/Libsnark/libsnark_abc-master/build/src
(enter) Call to alt_bn128_ate_precompute_G2 [ ] (1680490491.3708s x0.00 from start)
(leave) Call to alt_bn128_ate_precompute_G2 [0.0007s x0.98] (1680490491.3714s x0.00 from start)
(enter) Call to alt_bn128_ate_precompute_G2 [ ] (1680490491.3715s x0.00 from start)
(leave) Call to alt_bn128_ate_precompute_G2 [0.0006s x0.99] (1680490491.3721s x0.00 from start)
(enter) Call to r1cs_gg_ppzksnark_verifier_process_vk [0.0014s x0.96] (1680490491.3721s x0.00 from start)
(enter) Call to r1cs_gg_ppzksnark_online_verifier_strong_IC [ ] (1680490491.3721s x0.00 from start)
(enter) Call to r1cs_gg_ppzksnark_online_verifier_weak_IC [ ] (1680490491.3721s x0.00 from start)
(enter) Accumulate input [ ] (1680490491.3721s x0.00 from start)
(leave) Accumulate input [0.0000s x0.80] (1680490491.3722s x0.00 from start)
(enter) Check if the proof is well-formed [ ] (1680490491.3722s x0.00 from start)
(leave) Check if the proof is well-formed [0.0000s x0.68] (1680490491.3722s x0.00 from start)
(enter) Online pairing computations [ ] (1680490491.3722s x0.00 from start)
(enter) Check QAP divisibility [ ] (1680490491.3722s x0.00 from start)
(enter) Call to alt_bn128_ate_precompute_G1 [ ] (1680490491.3722s x0.00 from start)
(leave) Call to alt_bn128_ate_precompute_G1 [0.0000s x0.66] (1680490491.3723s x0.00 from start)
(enter) Call to alt_bn128_ate_precompute_G2 [ ] (1680490491.3723s x0.00 from start)
(leave) Call to alt_bn128_ate_precompute_G2 [0.0006s x0.99] (1680490491.3729s x0.00 from start)
(enter) Call to alt_bn128_ate_precompute_G1 [ ] (1680490491.3729s x0.00 from start)
(leave) Call to alt_bn128_ate_precompute_G1 [0.0000s x0.46] (1680490491.3730s x0.00 from start)
(enter) Call to alt_bn128_ate_precompute_G1 [ ] (1680490491.3730s x0.00 from start)
(leave) Call to alt_bn128_ate_precompute_G1 [0.0000s x0.45] (1680490491.3730s x0.00 from start)
(enter) Call to alt_bn128_ate_miller_loop [ ] (1680490491.3731s x0.00 from start)
(leave) Call to alt_bn128_ate_miller_loop [0.0014s x0.99] (1680490491.3744s x0.00 from start)
(enter) Call to alt_bn128_ate_double_miller_loop [ ] (1680490491.3745s x0.00 from start)
(leave) Call to alt_bn128_ate_double_miller_loop [0.0021s x1.00] (1680490491.3766s x0.00 from start)
(enter) Call to alt_bn128_final_exponentiation [ ] (1680490491.3766s x0.00 from start)
(enter) Call to alt_bn128_final_exponentiation_first_chunk [ ] (1680490491.3767s x0.00 from start)
(leave) Call to alt_bn128_final_exponentiation_first_chunk [0.0001s x0.90] (1680490491.3767s x0.00 from start)
(enter) Call to alt_bn128_final_exponentiation_last_chunk [ ] (1680490491.3767s x0.00 from start)
(leave) Call to alt_bn128_exp_by_neg_z [ ] (1680490491.3768s x0.00 from start)
(enter) Call to alt_bn128_exp_by_neg_z [0.0011s x0.99] (1680490491.3779s x0.00 from start)
(leave) Call to alt_bn128_exp_by_neg_z [ ] (1680490491.3780s x0.00 from start)
(enter) Call to alt_bn128_exp_by_neg_z [0.0005s x0.96] (1680490491.3784s x0.00 from start)
(leave) Call to alt_bn128_exp_by_neg_z [ ] (1680490491.3785s x0.00 from start)
(enter) Call to alt_bn128_exp_by_neg_z [0.0004s x0.97] (1680490491.3789s x0.00 from start)
(leave) Call to alt_bn128_final_exponentiation_last_chunk [0.0022s x0.95] (1680490491.3790s x0.00 from start)
(enter) Call to alt_bn128_final_exponentiation [0.0023s x0.93] (1680490491.3790s x0.00 from start)
(leave) Check QAP divisibility [0.0068s x0.95] (1680490491.3790s x0.00 from start)
(enter) Online pairing computations [0.0068s x0.94] (1680490491.3790s x0.00 from start)
(leave) Call to r1cs_gg_ppzksnark_online_verifier_weak_IC [0.0069s x0.94] (1680490491.3791s x0.00 from start)
(leave) Call to r1cs_gg_ppzksnark_online_verifier_strong_IC [0.0070s x0.94] (1680490491.3791s x0.00 from start)
(leave) Call to r1cs_gg_ppzksnark_verifier_strong_IC [0.0085s x0.93] (1680490491.3791s x0.00 from start)
验证结果:1
```

6 总结

通过本次实验,我对零知识证明进行了系统的学习,主要学习了交互式零知识证明和非交互式零知识证明,加深了理论知识的学习。我了解到,交互式零知识证明需要证明者和验证者时刻保持在线状态,而这会因网络延迟、拒绝服务等原因难以保障,而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。

在本次实验中,我们基于公共参考字符串 CRS 模型,实现了一个简单的简洁零知识证明 zkSNARK 的实例。通过例子的学习,我掌握了零知识证明的步骤,交互式证明协议流程和非交互零知识证明的步骤。此外,我还学习并使用 libsnark 的框架完成了实验,对零知识证明有了更深的理解。