南開大學
Nankai University

计算机学院和网络空间安全学院

数据安全实验报告

零知识证明实践

姓名：魏伯繁

学号：2011395

专业：信息安全

2023 年 4 月 11 日

目录

实验名称：零知识证明实践

1 实验要求

在本地搭建 libsnark 实验环境，跑通官方测试样例，复现课本样例，即验证 Alice 手中有方程：

$$x(x-1)(x-2)$$

的解，并通过阅读代码与注释自行编写程序，验证某人 Bob 拥有方程：

$$x^3+x+5=35$$

的解，编译运行，给出实验结果截图证明编写程序的正确性。

2 实验过程

2.1 实验背景知识

2.1.1 零知识证明

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。

举一个简单的例子，有一个缺口环形的长廊，出口和入口距离非常近（在目距之内），但走廊中间某处有一道只能用钥匙打开的门，A 要向 B 证明自己拥有该门的钥匙。采用零知识证明，则 B 看着 A 从入口进入走廊，然后又从出口走出走廊，这时 B 没有得到任何关于这个钥匙的信息，但是完全可以证明 A 拥有钥匙。

2.1.2 零知识

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。零知识证明应当具有正确性、完备性和零知识性三个性质

2.1.3 交互式 Schnorr 协议

Schnorr 机制是一种基于离散对数难题的零知识证明机制。证明者声称知道一个密钥 sk 的值，通过使用 Schnorr 加密技术，可以在不揭露 sk 的情况下，向验证者证明对 sk 的知情权。可用于证明你有一个私钥但是不披露私钥的内容。这是一类典型的可以应用到前面所述的身份认证场景的零知识证明协议。

依据椭圆曲线的离线对数难题，我们知道已知椭圆曲线 E 和生成元 G，随机选择一个整数 a，容易计算 Q=a*G，但是给定的 Q 和 G 计算 a 就非常困难。

承诺阶段：Alice 产生一个随机数 r，计算 R=r*G 并发送 R 给 Bob。
挑战阶段：Bob 提供一个随机数 c 进行挑战，并将 c 发送给 Alice。
回应挑战：Alice 根据挑战数 c 计算 z = r + a * c，然后把 z 发给 Bob。
验证阶段：Bob 通过式子进行检验：z*G ?= R + c*PK。

图 2.1: 协议流程

2.1.4 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，又叫 Fiat-Shamir Heurisitc（启发式），或者 Fiat-Shamir Paradigm（范式），由 Fiat 和 Shamir 在 1986 年提出，其特点是可以将交互式零知识证明转换为非交互式零知识证明，思路就是用公开的哈希函数的输出代替随机的挑战.

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)。
验证阶段：Bob(或者任意一个验证者)计算 c=Hash(PK,R)，验证 z*G?=R+c*PK。

图 2.2: 协议流程

为了不让 Alice 进行造假，在交互式 Schnorr 协议中需要 Bob 发送一个 c 值，并将 c 值构造进公式中。所以，在非交互式 Schnorr 协议中，如果 Alice 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，问题就解决了。生成这个公认无法造假的 c 的方法是使用随机数预言机。

2.1.5 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。zkSNARK 中比较典型的协议有 Groth10、GGPR13、Pinocchio、GRoth6、GKMMM18 等。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。它其实就把挑战过程中所要生成的随机数和挑战数，都预先生成好，然后基于这些随机数和挑战数生成他们对应的在证明和验证过程中所需用到的各种同态隐藏。之后，就把这些随机数和挑战数销毁。这些随机数和挑战数被称为 toxic waste（有毒废物），如果他们没有被销毁的话，就可以被用来伪造证明

zkSNARK 的命名几乎包含其所有技术特征：

- 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。

2.1.6 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。
- 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 libsnark/libsnark/zk_proof_systems 路径，就能发现 libsnark 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/ppzksnark/r1cs_ppzksnark 对应的是 BCTV14a
- zk_proof_systems/ppzksnark/r1cs_gg_ppzksnark 对应的是 Groth16

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。

2.2 实验环境配置

首先，按照教材的要求配置环境，具体内容在教材中均有说明，过程可以概括为在 github 上下载整个 libsnark 项目以及其依赖的 6 个子项目，在下载后一次编译安装六个子项目，运行测试完成后安装测试，在确实子项目安装无误后编译安装整体 libsnark 项目。

在完成上述环境配置后可以进入 libsnark_abc-master 后进行编译安装，在该目录下的 makelisttxt 中会将 src 目录下的 test.cpp 进行编译生成，随后使用./src/test 就可以执行目标程序，程序执行结果如下，说明环境配置完成。如图2.7所示

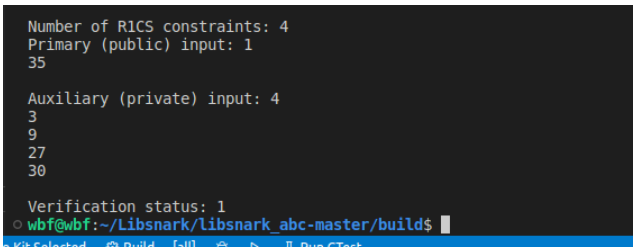

图 2.3: 官方样例跑通

2.3 修改代码完成要求

首先我们需要对 common.hpp 中的文件进行修改，为了验证

$$x^3 + x + 5 = 35$$

的正确性，我们依然使用五个中间变量 w_1-w_5 来完成从输入 x 到方程计算的任务，但是与样例不同的是，我们需要对五个中间变量进行重新赋值，赋值的流程如下，其中 w_3 用来表示 x 的三次

方，我们将原等式右侧的 35 移动到左侧构成常数项为-30 的一元三次方程式，中间变量 45 的构造过程如下面代码所示。

```
// pb.add_r1cs_constraint(r1cs_constraint<FieldT>(x, 1, x));
// x= w_1
pb.add_r1cs_constraint(r1cs_constraint<FieldT>(x, 1, w_1));
// x^2= w_2
pb.add_r1cs_constraint(r1cs_constraint<FieldT>(x, w_1, w_2));
// x^3= w_3
pb.add_r1cs_constraint(r1cs_constraint<FieldT>(x, w_2, w_3));
// x^3+x=w_4
pb.add_r1cs_constraint(r1cs_constraint<FieldT>(w_3+w_1, 1, w_4));
// x^3+x-30=w_5
pb.add_r1cs_constraint(r1cs_constraint<FieldT>(w_4-30, 1, w_5));
// 1*w_5=out
pb.add_r1cs_constraint(r1cs_constraint<FieldT>(w_5, 1, out));
```

在 myprove.cpp 中同样需要对 secret 数组进行修改，其对应的正是 common.hpp 中对中间变量的赋值过程，其中 x 代表自证者输入的秘密，随后对该秘密进行运算后得到结果。

```
//为私密输入提供具体数值
int secret[6];
secret[0]=x;
secret[1]=x;
secret[2]=x*x;
secret[3]=x*x*x;
secret[4]=x*x*x+x;
secret[5]=x*x*x+x-30;
```

接下来，我们检测代码的正确性。首先在 myprove 阶段输入 1,1 是错误答案，并不能让原方程式成立，随后运行 myverify 发现验证结果为 0，零知识证明验证失败。

```
wbf@wbf:~/Libsnark/libsnark_abc-master/build/src$ ./myprove
1
公有输入:1
0

私密输入:6
1
1
1
1
2
21888242871839275222246405745257275088548364400416034343698204186575808495589
```

图 2.4: 输入验证值

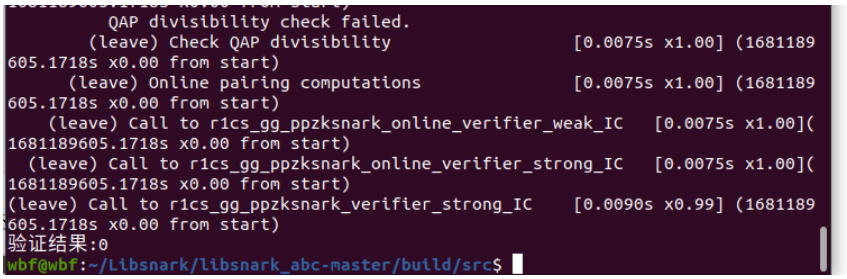

图 2.5: 查看验证结果

随后，我们检测算法是否能够检测出正确的答案，我们在 myprove 阶段输入 3,3 是正确答案，可以让原方程成立，随后运行 myverify 发现验证结果为 1，零知识证明验证成功

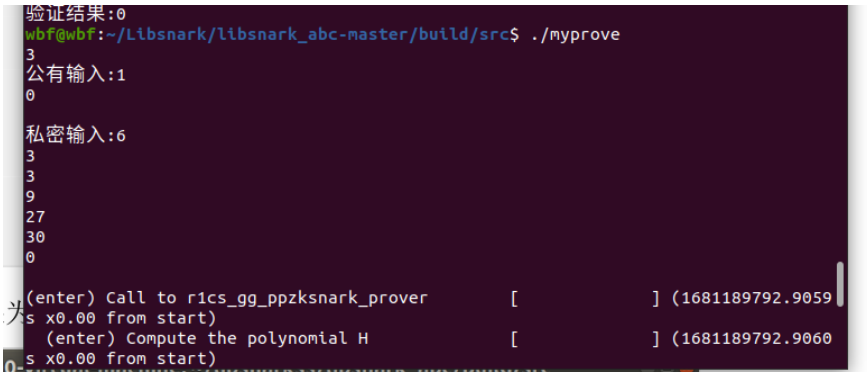

图 2.6: 输入验证值

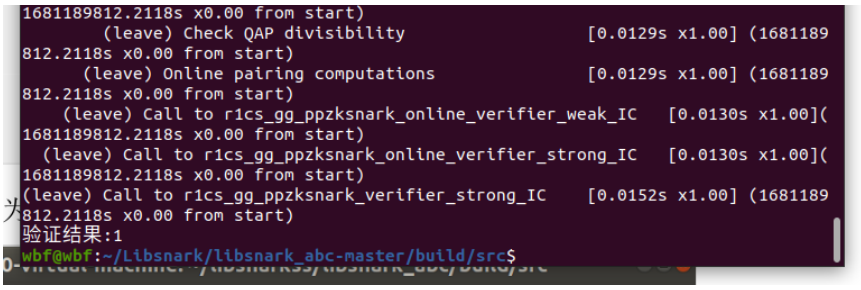

图 2.7: 查看验证结果

于是，我们正确验证了零知识证明验证正确和验证错误两种情况的代码运行结果，运行结果符合预期，代码编写成功。

3 实验心得

在进行数据安全的零知识证明实验之前，我对零知识证明的概念和原理并不是很清楚。但通过这次实验，我对于零知识证明的理解有了更深入的认识。在实验中，我们主要使用了与零知识证明相关的密码学技术，如 Hash 函数、椭圆曲线加密算法以及非对称加密算法。同时，我们还使用了基于 C 语言的 SNARK 和相关的库来实现相应的算法和操作。

在实验过程中，我们需要先选择一个证明对象，比如实验中的关于方程的解。然后，我们需要构建一个证明的过程，使得我们能够证明这个人掌握了方程的解但又不泄露解的值。这就需要我们运用到

零知识证明的特性，即证明者可以证明自己知道某个信息，但不需要向另一方透露该信息的具体内容。

在实验中，我们分别使用了 Schnorr 协议和 Pedersen Commitment Scheme 来实现零知识证明。通过使用这些算法和技术，我们实现了对数据的隐私保护，确保了证明对象的隐私信息得到了保护。

通过这次实验，我认识到零知识证明的重要性。在当前互联网时代，我们的个人隐私和数据安全面临着越来越大的挑战，而零知识证明可以作为一种强有力的数据隐私保护手段。同时，我也更深入地了解了密码学技术在数据安全中的重要性，比如 Hash 函数和加密算法的应用。

总之，这次实验让我对于数据安全和隐私保护有了更加深入的认识，提高了我的密码学技术应用能力，并且开拓了我的思考。