



南开大学
Nankai University

南 开 大 学

网络空间安全学院

数据安全实验报告

实验 4：零知识证明实践

姓名：付政烨

学号：2113203

年级：2021 级

专业：信息安全、法学

2024 年 4 月 23 日

目录

一、 实验要求	1
二、 实验内容	1
(一) 将待证明的命题表达为 R1CS (common.hpp)	1
(二) 生成证明密钥和验证密钥 (mysetup.cpp)	2
1. 密钥生成过程	2
2. 密钥存储	2
(三) 证明构造过程中的输入管理与证明文件生成 (myverify.cpp)	3
三、 实验结果	3
(一) CMakeLists.txt	3
(二) 结果展示	4
四、 实验心得与体会	5

一、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。

二、实验内容

(一) 将待证明的命题表达为 R1CS (common.hpp)

本部分核心在于利用 Rank-1 Constraint System (R1CS) 描述一个数学电路, 并在此基础上构建一个 protoboard。在示例中, 电路的功能被定义为 $C(V, W) = x^3 + x + 5$, 其中 V 表示公共输入, 而 W 代表私密输入。此电路的设计目的在于验证给定公共输入 V 的情况下, 电路的输出是否符合预期值 (例如输出为 35)。在零知识证明的框架下, 证明者需要验证他们知道一个符合条件的私密输入 W , 使得 $C(V, W) = 35$, 而无需直接透露 W 的值。

为实现此功能, 电路通过 R1CS 被转换为以下形式的一组等式: $w_1 = x \times x$ (计算 x 的平方) $w_2 = w_1 \times x$ (进一步计算 x 的立方) $w_3 = w_2 + x$ (将 x 的立方与 x 相加) $Out = w_3 + 5$ (最终添加常数 5 得到输出)

在零知识证明中, 构建 protoboard 为构造和验证证明提供了必要的结构。此结构不仅支持在证明阶段的构造, 还能在验证阶段确认证明的正确性。通过 protoboard, 证明者可以构造出满足 R1CS 约束的证明, 而验证者则可以检验这些证明是否符合约定的电路输出。使用 R1CS 对零知识证明的电路进行构建是在代码层面上实现的 (包含在 common.hpp 头文件中), 以支持零知识证明的初始化建立、论证和验证三个阶段。

在零知识证明中, protoboard 相当于是一个构造和处理电路约束的虚拟工作面, 用于存放和操作电路中的各种变量。首先, 我们定义了与电路相关的各个变量, 这包括公共输入、私密输入以及中间结果变量。这些变量通过 pb_variable 类在 protoboard 上进行分配。分配过程中使用 allocate() 函数具体来说, 这段代码包括以下几个关键步骤:

变量定义

```
1 pb_variable x; pb_variable w_1;  
2 pb_variable w_2;  
3 pb_variable w_3;  
4 pb_variable out;
```

变量 w_1 用于存放计算 $x * x$ 的结果; w_2 用于存放计算 $w_1 * x$ 的结果; w_3 用于存放计算 $w_2 + x$ 的结果; out 用于存放计算 $w_3 + 5$ 的最终结果。

变量与 protoboard 连接

```
1 out.allocate(pb, "out");  
2 x.allocate(pb, "x");  
3 w_1.allocate(pb, "w_1");  
4 w_2.allocate(pb, "w_2");  
5 w_3.allocate(pb, "w_3");
```

通过 allocate() 函数将这些变量连接到 protoboard 上, 每个变量通过其标识符能够在调试过程中被轻易地识别和跟踪。这样的操作确保了电路中的每个计算步骤都可以通过指定的变量来执

行和验证，形成了电路的基础结构，从而为后续的约束添加和证明生成提供了必要的基础。

在变量成功分配之后，我们需要定义它们之间的约束关系。这些关系通过 `add_r1cs_constraint()` 函数添加到 `protoboard` 中，每个约束对应电路中的一个特定计算步骤：

定义约束条件

```
1 pb.add_r1cs_constraint(r1cs_constraint<FieldT>(x, x, w_1));
2 pb.add_r1cs_constraint(r1cs_constraint<FieldT>(w_1, x, w_2));
3 pb.add_r1cs_constraint(r1cs_constraint<FieldT>(w_2+x, 1, w_3));
4 pb.add_r1cs_constraint(r1cs_constraint<FieldT>(w_3+5, 1, out));
```

首先，代码通过 R1CS 中的线性方程 $x \times x = w_1$ 实现表示输入 x 的平方等于中间变量 w_1 。然后，确保中间变量 w_1 （即 x 的平方）与 x 相乘的结果等于另一个中间变量 w_2 ，对应于计算 x 的立方。然后，用约束表示 w_2 （ x 的立方）与 x 相加的结果应等于 w_3 。最后这条约束确保从 w_3 加上常数 5 的结果存储在最终输出变量 `out` 中。这些约束共同形成了完整的电路逻辑，其中每个约束都是电路中一个具体的运算步骤。

将私密输入赋值给相应变量

```
1 if (secret!=NULL) {
2     pb.val(x)=secret[0];
3     pb.val(w_1)=secret[1];
4     pb.val(w_2)=secret[2];
5     pb.val(w_3)=secret[3];
6 }
```

`if (secret!=NULL) ...` 这部分代码用于在证明生成阶段将私密输入赋值给相应的变量。如果 `secret` 不为空，这意味着处在证明生成阶段，系统需要对输入变量进行赋值以满足约束条件。各个变量 x , w_1 , w_2 , 和 w_3 分别被赋予 `secret` 数组中的对应值，确保电路能够产生正确的输出。在这里，`secret` 数组包含了必要的私密输入，这些输入是外部不可见的，只在证明阶段内部使用，以确保在不透露具体输入的情况下验证计算的正确性。

在验证阶段，`secret` 将为 `NULL`，这表示不进行任何赋值操作，仅利用已经设置的约束来验证证明的正确性。这一阶段确保了证明者实际拥有满足约束的输入值，而无需直接查看这些值。

(二) 生成证明密钥和验证密钥 (mysetup.cpp)

1. 密钥生成过程

- **证明密钥**：此密钥用于生成证明，它包含了所有必要的信息和参数，使证明者能够根据给定的输入生成有效的证明。这可能包括加密参数、公式的哈希、和/或特定的配置信息，这些都是计算和构造证明所必需的。
- **验证密钥**：验证密钥用于验证由证明密钥生成的证明的正确性。它包含了执行验证算法所需的相关参数和指令，确保可以独立地验证证明的有效性，而不必重新执行原始计算。

2. 密钥存储

生成的证明密钥和验证密钥需要被保存至持久存储介质，以便在证明生成和验证过程中随时可用。通常，这些密钥以特定格式写入文件，文件格式应保证数据的完整性和可读性。存储密钥的文件可以使用加密或其他安全措施来保护密钥不被未授权访问。

在实际的操作中，这些密钥的生成是通过 `mysetup.cpp` 文件完成的。程序运行后，会自动输出证明密钥和验证密钥，并将它们分别保存到相应的文件中。证明密钥被存储以供证明者在构建其证明时使用；相对应地，验证密钥则被存储以供验证者在验证证明时使用，这部分代码不需要改动。

（三）证明构造过程中的输入管理与证明文件生成（`myverify.cpp`）

在形式化的证明系统中，证明者需要根据给定的公开输入和私有输入来构造证明。具体来说，公开输入在定义问题框架时已被赋予确定的数值，而私有输入则在证明构造阶段被指定。在这一过程中，证明者首先为私有输入设定具体的数值，然后将这些数值连同公开输入一起传递给证明函数（Prover function）。该函数负责生成相应的证明，这一证明随后会被保存至名为 `proof.raw` 的文件中，以便验证者（Verifier）之后进行验证。本部分只需要对 `myprove.cpp` 生成私密输入的地方进行修改：

`myverify.cpp`

```
1 int x;
2 cin >> x;
3 int secret [6];
4 secret [0]=x;
5 secret [1]=x*x;
6 secret [2]=x*x*x;
7 secret [3]=x*x*x*x;
```

注意，在 `common.hpp` 中的生成证明的部分，两者是一一对应的关系。

`common.hpp`

```
1 pb.val(x)=secret [0];
2 pb.val(w_1)=secret[1];
3 pb.val(w_2)=secret[2];
4 pb.val(w_3)=secret[3];
```

三、实验结果

（一）CMakeLists.txt

根据本次实验编写的 `CMakeLists.txt`。具体来说，首先通过 `include_directories` 函数将当前工作目录的路径添加到编译器的包含路径列表中，以确保编译器可以正确地找到所需的头文件。随后，通过 `add_executable` 函数声明了五个可执行文件 `main`、`test`、`range`、`mysetup` 和 `myprove`，分别用于编译生成相应的可执行程序。这些可执行文件在编译过程中与外部库 `snark` 进行链接，以便访问其提供的功能和资源。进一步地，通过 `target_link_libraries` 函数将这些可执行文件与 `snark` 库进行链接，以确保在运行时能够正确地调用其中定义的函数和符号。同时，通过 `target_include_directories` 函数将 `libsnaark` 和 `libqfft` 库的路径添加到包含路径中，以便编译器能够正确地找到这些库的头文件并进行相关的编译操作。最终，此代码段通过一系列针对每个可执行文件的类似操作，为每个文件设置了相应的链接和包含路径，以确保编译过程中的依赖关系得到满足，并最终生成可执行程序。

接着，运行如下命令：

```

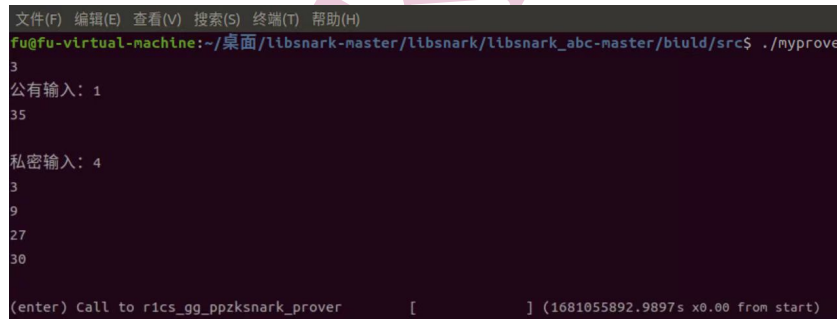
1 cmake ..
2 make
3 cd src
4 ./mysetup
5 ./myprove
6 3
7 ./myverify

```

(二) 结果展示

在零知识证明框架的实验中，设定：“公有输入 =1”和“私密输入 =4”。含义如下所示：

- **公有输入 (pb.primary_input())**: 这一部分代表零知识证明中的主要输入，即需要证明正确性的语句的输入参数。在零知识证明的环境中，目标是证实某个命题的正确性而不透露命题的详细内容。简而言之，公有输入就是这种需要验证正确性的命题的输入参数。公有输入被设定为 35，意味着需要验证输出值为 35 的命题。
- **私密输入 (pb.auxiliary_input())**: 这一部分是零知识证明中的辅助输入，通常保持私密。在这种框架下，证明者需要证实他知晓一组特定的私有输入值，这些值能够满足某个预设的命题，同时又不公开这些私有输入的具体内容。在具体实现中，可能存在多个这样的私有输入（4 个），包括一些中间变量，它们共同构成满足某个命题的证明基础。

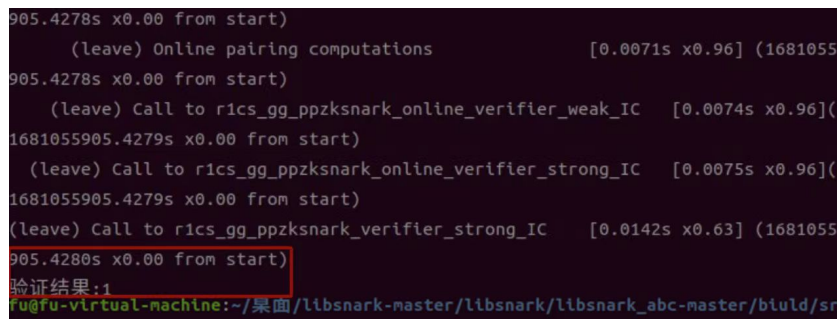


```

文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
fu@fu-virtual-machine:~/桌面/libsark-master/libsark/libsark_abc-master/build/src$ ./myprove
3
公有输入: 1
35
私密输入: 4
3
9
27
30
(enter) Call to r1cs_gg_ppzksnark_prover [ ] (1681055892.9897s x0.00 from start)

```

图 1



```

905.4278s x0.00 from start)
  (leave) Online pairing computations [0.0071s x0.96] (1681055
905.4278s x0.00 from start)
  (leave) Call to r1cs_gg_ppzksnark_online_verifier_weak_IC [0.0074s x0.96](
1681055905.4279s x0.00 from start)
  (leave) Call to r1cs_gg_ppzksnark_online_verifier_strong_IC [0.0075s x0.96](
1681055905.4279s x0.00 from start)
  (leave) Call to r1cs_gg_ppzksnark_verifier_strong_IC [0.0142s x0.63] (1681055
905.4280s x0.00 from start)
验证结果:1
fu@fu-virtual-machine:~/桌面/libsark-master/libsark/libsark_abc-master/build/src$

```

图 2

在实验中观察到，结果显示为 1，表明通过了验证步骤，验证成功！

四、 实验心得与体会

本次实验通过使用 libsnark 库并基于 R1CS (Rank-1 Constraint System) 实现了实验要求的示例。R1CS 约束系统作为描述复杂关系的强大工具，其核心是通过一组公有变量和一组私有变量构成的约束关系，展现出了数据结构的高效性。在这一过程中，我首次接触到了 proto-board，这一数据结构帮助我定义和管理变量及其约束，是我能成功构建证明的关键。实验中使用的 r1cs_gg_ppzksnark 算法基于双线性对，不仅提升了零知识证明的生成效率，还保障了安全性。通过这次学习，我不仅掌握了如何操作 libsnark 的关键函数，包括加载密钥、生成和验证证明，也逐步克服了环境配置的挑战。原本在 Ubuntu 20.04 下因默认编译问题导致的障碍，在切换至 C++11 编译后得以解决。

尽管安装 libsnark 库的过程异常艰难，这次实验的成功不仅增强了我对零知识证明技术的理解，也锻炼了我的编程和问题解决能力。通过这些技术的探索，我对数据安全的理解更加深刻，相信这将为我未来的学习和工作带来不小的助力。

MINI