



南開大學
Nankai University

南 開 大 學

計 算 機 學 院

區塊鏈課程實驗報告

Ex6

信息安全 2211985 李佳璐

計算機科學與技術 2211290 姚知言

2024 年 12 月 13 日

目录

| | |
|---|-----------|
| 一、 实验过程 | 1 |
| (一) 环境配置 | 1 |
| (二) 了解 circom | 1 |
| 1. 阅读 circuits/example.circom 中的电路示例 | 1 |
| 2. 使用 SmallOddFactorization 电路为 $7 \times 17 \times 19 = 2261$ 创建证明 | 2 |
| (三) 开关电路 | 4 |
| 1. IfThenElse 函数 | 4 |
| 2. SelectiveSwitch 函数 | 5 |
| (四) 消费电路 | 6 |
| (五) 计算花费电路的输入 | 7 |
| (六) 赎回证明 | 8 |
| (七) 测试 | 9 |
| 二、 问题解答 | 10 |
| (一) Question 1 | 10 |
| (二) Question 2 | 10 |
| (三) Question 3 | 10 |

一、实验过程

(一) 环境配置

在 Ubuntu 虚拟机中，运行以下指令，完成 nodejs,npm,snarkjs,circom,mocha 的安装。

```
1 sudo apt install nodejs npm
2 bash install.sh
```

运行 `npm test`，部分样例通过，当前环境已经没有问题。

```

npm notice New patch version of npm available! 10.9.0 -> 10.9.2
npm notice Changelog: https://github.com/npm/cli/releases/tag/v10.9.2
npm notice To update run: npm install -g npm@10.9.2
npm notice
~/Desktop/区块链作业Ex6补充材料/Ex6_包含全部依赖库/Ex6 > npm test

> cs251-cash@0.1.0 test
> mocha -s is -t 5s test/IfThenElse.js test/selective_switch.js test/compute_spend_inputs.js test/spend.js

IfThenElse
  1) should give 'false_value' when 'condition' = 0
  2) should give 'true_value' when 'condition' = 1
  3) should enforce that s in {0, 1}

SelectiveSwitch
  4) should not switch when s = 0
  5) should switch when s = 1
  6) should enforce that s in {0, 1}

computeInput
  7) transcript0.txt, depth 0, nullifier 1
  8) transcript1.txt, depth 4, nullifier 4
  9) transcript2.txt, depth 25, nullifier 7

Spend
  ✓ witness computable for depth 0
  ✓ witness computable for depth 1
  ✓ witness computable for depth 2
  10) witness not computable for bad input

3 passing (89ms)
10 failing

1) IfThenElse
   should give 'false_value' when 'condition' = 0:
     Error: Signal not assigned: main.out
       at calculateWitness (node_modules/snarkjs/src/calculateWitness.js:63:19)
       at Circuit.calculateWitness (node_modules/snarkjs/src/circuit.js:59:16)
       at Context.<anonymous> (test/IfThenElse.js:25:30)
       at callFn (mocha/lib/runnable.js:372:21)
       at Runnable.run (mocha/lib/runnable.js:364:7)

```

图 1: 环境配置

(二) 了解 circom

1. 阅读 circuits/example.circom 中的电路示例

回答 artifacts/writeup.md 中的相关问题在该报告未解答，并在相应文件中已更新。

- Num2Bits 电路

这个电路的目的是将输入信号 `in` 转换为一个二进制位数组 `bits`，其中 `bits[0]` 是最低有效位，`bits[b-1]` 是最高有效位。

将输入 `in` 右移 `i` 位并与 1 进行按位与运算，从而提取出 `in` 的每一位：

```
1 $bits[i] <— (in >> i) & 1;$
```

约束每一位为 0 或 1：

```
1 $bits[i] * (1 - bits[i]) == 0;$
```

构造和约束 `in` 的二进制表示的和：

```

1 var sum_of_bits = 0;
2 for (var i = 0; i < b; ++i) {
3     sum_of_bits += (2 ** i) * bits[i];
4 }

```

```
5 sum_of_bits == in;
```

这部分代码将 bits 数组中的每一位按权重相加,构成一个线性组合,最终通过约束 `sum_of_bits == in` 强制 `sum_of_bits` 等于原始输入值 `in`, 确保每个 `bits[i]` 的二进制表示是正确的。

- SmallOdd 电路

这个电路的功能是强制 `in` 是一个小于 2^b 的奇数。

首先使用了 Num2Bits 电路将 `in` 转换为二进制位数组 `bits`。

强制最低有效位为 1:

```
1 binaryDecomposition.bits[0] == 1;
```

- SmallOddFactorization 电路

这个电路的功能是将输入信号 `product` 分解为 `n` 个小于 2^b 的奇数因子。

使用 SmallOdd 子电路来确保每个因子是一个小于 2^b 的奇数:

```
1 component smallOdd[n];
2 for (var i = 0; i < n; ++i) {
3     smallOdd[i] = SmallOdd(b);
4     smallOdd[i].in <= factors[i];
5 }
```

由于有多个因子需要相乘,因此使用了辅助信号 `partialProducts` 来分步计算乘积,并将每一步的乘积与下一个因子相乘,直到计算出完整的乘积:

```
1 signal partialProducts[n + 1];
2 partialProducts[0] <= 1;
3 for (var i = 0; i < n; ++i) {
4     partialProducts[i + 1] <= partialProducts[i] * factors[i];
5 }
6 product == partialProducts[n];
```

- main 电路

main 电路调用 **SmallOddFactorization 电路**, 要求将一个输入 `product` 分解为 3 个小于 2^8 的奇数因子。

```
1 component main = SmallOddFactorization(3, 8);
```

2. 使用 SmallOddFactorization 电路为 $7 \times 17 \times 19 = 2261$ 创建证明

1. 编译 circom 电路, 将其生成 json 文件

```
1 circom example.circom
```

该命令会对 `example.circom` 文件进行编译, 将编译结果保存为 `circuit.json`。

2. 创建可信设置

```
1 snarkjs setup
```

在 *snarkjs* 中, *setup* 主要用于完成可信初始化, 并生成系统所需的公共参数和密钥对。该步骤借助 *circuit.json* 生成 *proving_key.json* 和 *verification_key.json*。

其中:

- *proving_key.json* 是证明者所需的密钥, 用于生成零知识证明;
- *verification_key.json* 是验证者所需的密钥, 用于验证证明的有效性。

3. 创建 input.json

input.json 文件用于提供零知识证明所需的输入数据, 包含电路运行所需的公开输入和私有输入的值。

以下展示了 *example.circom* 中, *SmallOddFactorization* 电路的输入:

```
1 template SmallOddFactorization(n, b) {  
2     signal input product;  
3     signal private input factors[n];  
4     ...  
5 }  
6 component main = SmallOddFactorization(3, 8);
```

输入数据包括:

- product: 规定的输入乘积。
- factor: 私有输入, 包括 3 个因子。

根据输入和题目要求编写的 *input.json*:

```
1 {  
2   "product": 2261,  
3   "factors": [7,17,19]  
4 }
```

4. 创建 witness

witness 是零知识证明生成过程中重要的中间数据, 包含了满足电路约束所需的所有中间变量和输出信号的值。

```
1 snarkjs calculatewitness
```

这一命令可以根据 *circuit.json*, 以及 *input.json* 的输入计算并生成 *witness.json*。

5. 创建证明

```
1 snarkjs proof
```

此命令会默认使用 *proving_key.json* 和 *witness.json* 文件生成 *proof.json* 和 *public.json*。

- *proof.json* 文件包含实际的证明。
- *public.json* 文件包含公开输入和输出的值。

6. 验证证明

```
1 snarkjs verify
```

此命令会使用 *verification_key.json*, *proof.json* 和 *public.json* 验证证明是否有效。即验证我们知道一个 *witness*，其中公共输入和输出与 *public.json* 文件相匹配。验证成功会显示 *OK*，验证失败会显示 *INVALID*。

全过程命令如图2所示。

```
~/De/区/E/Ex6/circuits } circom example.circom
~/De/区/E/Ex6/circuits } snarkjs setup
~/De/区/E/Ex6/circuits } vim input.json
~/De/区/E/Ex6/circuits } snarkjs calculatewitness
~/De/区/E/Ex6/circuits } snarkjs proof
~/De/区/E/Ex6/circuits } snarkjs verify
OK
```

图 2: 使用 SmallOddFactorization 电路创建证明

按照要求将验证密钥 (*verifierkey*) 保存到 *artifacts/verifier_key_factor.json* 中，将证明保存到 *artifacts/proof_factor.json* 中。

(三) 开关电路

1. IfThenElse 函数

IfThenElse 电路验证了条件表达式的正确求值。它有 1 个输出，和 3 个输入：

- *condition*: 应该是 0 或 1
- *true_value*: 如果 *condition* 是 1, 那么输出 *true_value*
- *false_value*: 如果 *condition* 是 0, 那么输出 *false_value*

```
1 template IfThenElse() {
2     signal input condition;
3     signal input true_value;
4     signal input false_value;
5     signal output out;
6
7     // TODO
8     // Hint: You will need a helper signal...
9     condition * (1 - condition) == 0; // 保证 condition 为 0 或 1
10    condition * (1 - condition) == 0; // 保证 condition 为 0 或 1
11
12    // 使用 helper signal
13    signal true_condition;
14    signal false_condition;
15
16    // 构建条件选择
17    true_condition <== condition * true_value;
```

```

18     false_condition <== (1 - condition) * false_value;
19
20     // 最终结果通过线性组合得到
21     out <== true_condition + false_condition;
22 }

```

- 执行约束条件，确保 condition 是 0 或 1。
- 如果条件为真，将 true_condition 设置为 condition * true_value。
- 如果条件为假，将 false_condition 设置为 (1 - condition) * false_value。
- 将 out 设置为 true_condition 和 false_condition 的和，实现条件语句。

在 circom 中，由于只能使用乘法和加法等基本运算，如果直接使用 true_value 和 condition 来计算 false_value 的线性组合，将无法实现条件语句的效果。因此引入了辅助的 signal。

2. SelectiveSwitch 函数

SelectiveSwitch 电路根据输入信号 s 的值选择性地切换输入信号 in0 和 in1。

- 如果 s 为 1，则 out0 将等于 in1，out1 将等于 in0
- 如果 s 为 0，则 out0 将等于 in0，out1 将等于 in1

通过两个 IfThenElse 电路，选择不同的输入，分别完成 out0 和 out1 的输出计算，来实现 SelectiveSwitch 电路：

```

1 template SelectiveSwitch() {
2     signal input in0;
3     signal input in1;
4     signal input s;
5     signal output out0;
6     signal output out1;
7
8     // TODO
9     // 使用 IfThenElse 组件来选择不同的输入
10    component ifthenelse0 = IfThenElse();
11    component ifthenelse1 = IfThenElse();
12
13    // 配置第一个 IfThenElse 组件
14    ifthenelse0.condition <== s;
15    ifthenelse0.true_value <== in1;
16    ifthenelse0.false_value <== in0;
17    out0 <== ifthenelse0.out; // 将输出连接到 out0
18
19    // 配置第二个 IfThenElse 组件
20    ifthenelse1.condition <== s;
21    ifthenelse1.true_value <== in0;
22    ifthenelse1.false_value <== in1;
23    out1 <== ifthenelse1.out; // 将输出连接到 out1
24 }

```

(四) 消费电路

Spend 电路用于验证在深度为 depth 的 Merkle 树中，树根为 digest，是否存在 H(nullifier, nonce)。

- 这个存在性由 Merkle 证明提供，额外的输入为 sibling 和 direction
- sibling[i]：在路径上到达此硬币的节点的第 i 层时的兄弟节点。
- direction[i]：“0”或“1”，指示该兄弟节点在左边还是右边。
- sibling 哈希直接对应于 SparseMerkleTree 路径中的兄弟节点。
- direction 将布尔方向从 SparseMerkleTree 路径转换为字符串表示的整数（“0”或“1”）。

```

1  template Spend(depth) {
2      // 公共输入：Merkle 树的根节点哈希和 nullifier
3      signal input digest;
4      signal input nullifier;
5
6      // 私有输入：nonce、Merkle 路径中的 sibling 节点哈希值和方向
7      signal private input nonce;
8      signal private input sibling[depth];
9      signal private input direction[depth];
10
11     // 中间变量存储每一层的哈希值
12     signal MerkleTree[depth + 1];
13
14     // 第 0 层：计算硬币的哈希值 coin = H(nullifier, nonce)
15     component coinHash = Mime2();
16     coinHash.in0 <== nullifier;
17     coinHash.in1 <== nonce;
18     MerkleTree[0] <== coinHash.out;
19
20     // 遍历每一层，处理 Merkle 路径
21     component hash[depth]; // 哈希组件数组
22     component switch[depth]; // 选择开关组件数组
23
24     for (var i = 0; i < depth; i++) {
25         // 初始化选择开关，用于根据方向选择左右子节点
26         switch[i] = SelectiveSwitch();
27         switch[i].in0 <== MerkleTree[i]; // 当前哈希值
28         switch[i].in1 <== sibling[i]; // 当前层的兄弟节点哈希值
29         switch[i].s <== direction[i]; // 方向：0 表示左节点，1 表示
           右节点
30
31         // 初始化 Mime2 哈希，用于计算下一层的哈希值
32         hash[i] = Mime2();
33         hash[i].in0 <== switch[i].out0; // 左节点
34         hash[i].in1 <== switch[i].out1; // 右节点

```



```
35     MerkleTree[i + 1] <= hash[i].out;    // 计算得到的下一层哈希值
36 }
37
38 // 最后一层的哈希值 MerkleTree[depth] 必须等于提供的根节点哈希 digest
39 MerkleTree[depth] == digest;
40 }
```

(五) 计算花费电路的输入

computeInput 函数用于计算 Spend 电路的输入参数。

输入:

- depth : 正在使用的 Merkle 树的深度。
- transcript : 一个包含所有添加到树中的硬币的列表。每个项都是一个数组, 如果数组只有一个元素, 则该元素是一个具有单一价值的 coin。否则, 数组将有两个元素, 按顺序为: nullifier 和 nonce。此列表不包含任何重复的 nullifiers 或 coins。
- nullifier : 要为其生成验证器输入的 nullifier。此 nullifier 将是 transcript 中的 nullifiers 之一。

```
1 function computeInput(depth, transcript, nullifier) {
2     // 初始化 Merkle 树
3     var tree = new SparseMerkleTree(depth);
4     var nonce = null;
5
6     // 将 transcript 编译成树并查找 nullifier
7     for (var i = 0; i < transcript.length; i++) {
8         if (transcript[i].length == 1) {
9             // 如果元素个数为1, 直接插入树中
10            tree.insert(transcript[i]);
11        } else if (transcript[i].length == 2) {
12            // 如果元素个数为2, 检查是不是 nullifier
13            if (transcript[i][0] == nullifier) {
14                nonce = transcript[i][1];
15            }
16            // 哈希后添加到树中
17            tree.insert(mimc2(transcript[i][0], transcript[i][1]));
18        } else {
19            throw new Error("读取 transcript 时出现问题");
20        }
21    }
22
23    // 检查是否找到 nullifier
24    if (nonce == null) {
25        throw new Error("找不到 nullifier");
26    }
27
28    // 构造输入对象
```

```

29     var computedInput = {
30         digest: tree.digest,
31         nullifier: nullifier,
32         nonce: nonce
33     };
34
35     // 计算 Merkle 路径
36     var path = tree.path(mimc2(nullifier, nonce));
37     for(let i = 0; i < path.length; ++i) {
38         const [sibling, direction] = path[i];
39         var sibling_string = "sibling[" + i + "]";
40         computedInput[sibling_string] = sibling;
41         var direction_string = "direction[" + i + "]";
42         computedInput[direction_string] = (0 + direction).toString();
43     }
44
45     return computedInput;
46 }

```

整体实现流程如下：

1. **Merkle 树构建：**创建一个具有指定深度的 SparseMerkleTree 对象。

2. **Transcript 处理：**遍历给定的 transcript，它是一个记录硬币信息的数组。每个记录可以是一个包含硬币的数组，或者包含两个元素的数组，分别是 nullifier 和 nonce。

- 如果记录只包含一个元素，直接将该元素插入 Merkle 树中。
- 如果记录包含两个元素，检查是否为要验证的 nullifier，如果是，则将其对应的 nonce 记录下来，并将哈希后的结果插入 Merkle 树中。

3. **输入参数计算：**构建一个包含 Spend 电路验证所需输入参数的对象，其中包括：digest：transcript 应用后整个树的 digest。nullifier：正在花费的硬币的 nullifier。nonce：该硬币的 nonce。sibling[i]：在路径上到达此硬币的节点的第 i 层时的兄弟节点。direction[i]：“0”或“1”，指示该兄弟节点在左边还是右边。

4. **路径计算：**通过调用 Merkle 树的 path 方法获取到达指定 nullifier 的 Merkle 路径，然后将路径中的每个兄弟节点和方向信息添加到输入参数对象中。

(六) 赎回证明

1. 编译 circom 电路，将其生成为 json 文件

```
1 circom ../test/circuits/spend10.circom
```

该命令会对 spend10.circom 文件进行编译，将编译结果保存为 circuit.json。

2. 创建可信设置

```
1 snarkjs setup
```

该命令借助 circuit.json 生成 proving_key.json 和 verification_key.json。

3. 创建 input.json

根据输入和题目要求调用 `compute_spend_inputs.js`，按照实验要求传入对应参数，生成 `input.json`：

```
1 node ../src/compute_spend_inputs.js -o input.json 10 ../test/
  compute_spend_inputs/transcript3.txt 10137284576094
```

4. 创建 witness

```
1 snarkjs calculatewitness
```

这一命令根据 `circuit.json`，以及 `input.json` 的输入计算并生成 `witness.json`。

5. 创建证明

```
1 snarkjs proof
```

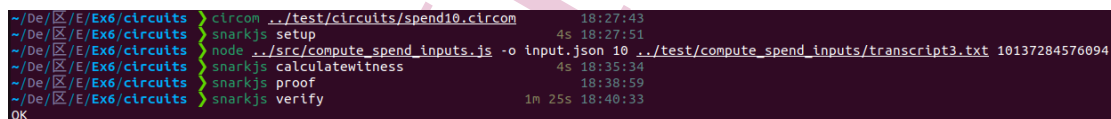
此命令默认使用 `proving_key.json` 和 `witness.json` 文件生成 `proof.json` 和 `public.json`。

6. 验证证明

```
1 snarkjs verify
```

此命令会使用 `verification_key.json`、`proof.json` 和 `public.json` 验证证明是否有效。期待的输出是 `OK`。

全过程命令如图3所示。



```
~/De/Ex6/circuits > circom ../test/circuits/spend0.circom 18:27:43
~/De/Ex6/circuits > snarkjs setup 4s 18:27:51
~/De/Ex6/circuits > node ../src/compute_spend_inputs.js -o input.json 10 ../test/compute_spend_inputs/transcript3.txt 10137284576094
~/De/Ex6/circuits > snarkjs calculatewitness 4s 18:35:34
~/De/Ex6/circuits > snarkjs proof 18:38:59
~/De/Ex6/circuits > snarkjs verify 1m 25s 18:40:33
OK
```

图 3: 赎回证明

输出了 `OK`，赎回证明验证成功。

根据要求，将验证密钥 (`verifier key`) 保存到 `artifacts/verifier_key_spend.json` 中，证明保存到 `artifacts/proof_spend.json` 中。

(七) 测试

运行 `npm test`，全部模块的测试均顺利通过，如图4所示，实验完成。

```
~/Desktop/区块链作业Ex6补充材料/Ex6_包含全部依赖库/Ex6 > npm test

> cs251-cash@0.1.0 test
> mocha -s 1s -t 5s test/if_then_else.js test/selective_switch.js test/compute_spend_inputs.js test/spend.js

IfThenElse
  ✓ should give 'false_value' when 'condition' = 0
  ✓ should give 'true_value' when 'condition' = 1
  ✓ should enforce that s in {0, 1}

SelectiveSwitch
  ✓ should not switch when s = 0
  ✓ should switch when s = 1
  ✓ should enforce that s in {0, 1}

computeInput
  ✓ transcript0.txt, depth 0, nullifier 1
  ✓ transcript1.txt, depth 4, nullifier 4
  ✓ transcript2.txt, depth 25, nullifier 7

Spend
  ✓ witness computable for depth 0
  ✓ witness computable for depth 1
  ✓ witness computable for depth 2 (2110ms)
  ✓ witness not computable for bad input (1164ms)

13 passing (4s)
```

图 4: 实验代码验证

二、 问题解答

以下是 *artifacts/writeup.md* 中问题的解答:

(一) Question 1

Question: 在下面 *Num2Bits* 代码片段中, 看起来 *sum_of_bits* 可能是一个信号的乘积和, 使得随后的约束不是一级的。说明为什么 *sum_of_bits* 实际上是一个信号的**线性组合**。

```
1 sum_of_bits += (2 ** i) * bits[i];
```

Answer: *sum_of_bits* 是一个信号的**线性组合**, 因为和里的每一项都由一个定值 (二的幂, 2^i) 与一个信号 (*bits[i]*) 乘等构成。在电路限制中, 线性组合指的是每项都由定值与信号相乘后相加构成。由于 2^i 是在编译时定义的幂值, 不依赖于信号的值, 因此这个和满足线性组合的要求。它没有包含两个信号乘等, 故而不是非线性的。

(二) Question 2

Question: 用你自己的言语说明 `<==` 这个运算符的意义。

Answer: `<==` 运算符在 *circom* 中用于定义两个信号或表达式之间的相等约束。它确保表达式左边和右边在执行时的计算结果相等。这个运算符不会为左边的信号赋值; 而是创建一个约束, 需要在电路验证中满足才能被认为有效。

这与 `<--` 运算符有所不同, `<--` 用于为信号赋值, 但不创建任何约束。

(三) Question 3

Question: 假设你正在阅读一个 *circom* 程序, 你看到下列内容:

```
1 signal input a;
2 signal input b;
```

```
3   signal input c;  
4   (a & 1) * b == c;
```

说明为什么这段代码是无效的。

Answer: 这个约束是无效的，因为 `&` 操作符是按位与运算，它是位运算，而在 `circom` 中，电路约束必须使用算术运算，即加法和乘法等线性操作才可以用来建立约束。因此，位运算（如 `&`）不能直接在 `circom` 电路中使用。

为了使这个约束有效，可以使用适当的算术操作来代替位运算：

```
1   (a % 2) * b == c;
```

在 `circom` 中，`a % 2` 是可以用来代替 `a & 1` 的，因为它们在判断奇偶性时的结果是相同的。

这行代码通过将 `a` 除以 2 后的余数来模拟按位与操作，这样就变成了一个有效的算术约束，适合在 `circom` 中使用。

MINIB