区块链第六次作业

> 2111460 张洋 2111617 尚然

一、了解circum

1. 分析 `circuits/example.circom` 中的电路

一个 circom 程序实际上做两件事：

- 对信号的约束
- 计算非输入信号

`Num2Bits` 中实现了将 `in` 分解为由 `bits` 给出的 `b` 位，`bit[0]` 是最低有效位。

```
template Num2Bits(b) {
    signal input in;
    signal output bits[b];
    // 计算位的值
    for (var i = 0; i < b; ++i) {
        // 使用 `<--` 为信号分配值而不对其进行约束。
        // 虽然约束只能使用乘法/加法，但赋值可以使用任何操作。
        bits[i] <-- (in >> i) & 1;
    }
    // 每个位约束为 0 或 1。
    for (var i = 0; i < b; ++i) {
        // 使用 `===` 强制执行一阶约束（R1C）。
        bits[i] * (1 - bits[i]) === 0;
     // ^--A--^   ^-----B-----^     C
     // 在这个 R1C 中的线性组合 A、B 和 C。
    }
    // 此 var 是一个线性组合。
    var sum_of_bits = 0;
    for (var i = 0; i < b; ++i) {
        sum_of_bits += (2 ** i) * bits[i];
    }
    // 将这个和（是信号的线性组合）约束为 in。
    sum_of_bits === in;
}
```

`SmallOdd` 中实现了强制执行 `in` 是小于 2^b 的奇数。

```
template SmallOdd(b) {
    signal input in;
    // 声明和初始化一个子电路;
    component binaryDecomposition = Num2Bits(b);
    // 使用 <== 同时分配和约束。
    binaryDecomposition.in <== in;
    // 约束最低有效位为 1。
    binaryDecomposition.bits[0] === 1;
}
```

`SmallOddFactorization`强制执行 `product` 的因式分解为 `n` 个小于 2^b 的奇数因子。

```
template SmallOddFactorization(n, b) {
    signal input product;
    signal private input factors[n];
    // 将每个因子约束为小而奇。
    // 我们将需要 n 个子电路来进行奇性检查。
    component smallOdd[n];
    for (var i = 0; i < n; ++i) {
        smallOdd[i] = SmallOdd(b);
        smallOdd[i].in <== factors[i];
    }
    // 现在将因子约束为乘积。由于有许多乘法，我们将乘法拆分为 R1C。
    signal partialProducts[n + 1];
    partialProducts[0] <== 1;
    for (var i = 0; i < n; ++i) {
        partialProducts[i + 1] <== partialProducts[i] * factors[i];
    }
    product === partialProducts[n];
}
```

最后为此文件设置 `main` 电路，这是 `circom` 将合成的电路。

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

相关问题的答案已保存到`artifacts/writeup.md`中。

2. 使用 `SmallOddFactorization` 电路为7×17×19=2261创建一个证明

1. 编译`example.circom`，把结果保存到`example.json`。

```
z@z-virtual-machine:~/Documents/Ex6 (1)/circuits$ circom example.circom -o example.json
```

2. `snarkjs` 将查找并使用 `example.json` 进行设置。

```
z@z-virtual-machine:~/Documents/Ex6 (1)/circuits$ snarkjs setup -c example.json
```

3. 创建一个名为 `input.json` 的文件。

```
{
    "product": 2261,
    "factors": [7, 17, 19]
}
```

4. 计算witness并保存到 `witness.json` 文件中。

```
z@z-virtual-machine:~/Documents/Ex6 (1)/circuits$ snarkjs calculatewitness -c example.json
```

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

```
z@z-virtual-machine:~/Documents/Ex6 (1)/circuits$ snarkjs proof
```

   - `proof.json` 文件将包含实际的证明。

     按照要求保存到了 `artifacts/proof_factor.json` 中。

   - `public.json` 文件将仅包含公共输入和输出的值。

6. 验证 `proof`，命令使用 `verification_key.json`、`proof.json` 和 `public.json` 进行验证，以确保其有效。在这里，我们正在验证我们知道一个 `witness`，其中公共输入和输出与 `public.json` 文件中的输入相匹配。如果证明有效，输出 `OK`，如果无效，则为 `INVALID`。

```
z@z-virtual-machine:~/Documents/Ex6 (1)/circuits$ snarkjs verify
OK
```

验证密钥（verifier key）保存到了 `artifacts/verifier_key_factor.json` 中，证明保存到了 `artifacts/proof_factor.json` 中。

二、开关电路

1. IfThenElse

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

```
template IfThenElse() {
    signal input condition;
    signal input true_value;
    signal input false_value;
    signal output out;

    // TODO
    // Hint: You will need a helper signal...
    condition * (1 - condition) === 0;
    signal true_condition;
    signal false_condition;
    true_condition <== condition * true_value;
    false_condition <== (1 - condition) * false_value;
    out <== true_condition + false_condition;
}
```

执行约束条件，确保 `condition` 是 0 或 1。使用辅助的 `signal` 和乘法操作来实现 `IfThenElse` 逻辑。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 `condition` 来计算 `true_value` 和 `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

`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`。

利用 `IfThenElse` 电路来实现 `SelectiveSwitch` 电路。

```
template SelectiveSwitch() {
    signal input in0;
    signal input in1;
    signal input s;
    signal output out0;
    signal output out1;

    // TODO
    component ifthenelse0 = IfThenElse();
    component ifthenelse1 = IfThenElse();

    ifthenelse0.condition <== s;
    ifthenelse0.true_value <== in1;
    ifthenelse0.false_value <== in0;
    out0 <== ifthenelse0.out;

    ifthenelse1.condition <== s;
    ifthenelse1.true_value <== in0;
    ifthenelse1.false_value <== in1;
    out1 <== ifthenelse1.out;
}
```

`SelectiveSwitch` 电路使用了两个 `IfThenElse` 组件，根据输入信号 `s` 的值有条件地选择性地切换输入信号 `in0` 和 `in1`。

三、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"）。

对于用于确定硬币和 `Merkle` 树的哈希函数 H，使用哈希函数 `Mimc2`，它有 1 个输出，和 2 个输入。

```
template Spend(depth) {
    signal input digest;
    signal input nullifier;
    signal private input nonce;
    signal private input sibling[depth];
    signal private input direction[depth];

    // TODO
    // 需要 depth+1 个中间路径哈希用于 Merkle 路径
    signal MerkleTree[depth+1];

    // coinHash是MerkleTree的第一个输入
    component coinHash = Mimc2();
    coinHash.in0 <== nullifier;
    coinHash.in1 <== nonce;
    MerkleTree[0] <== coinHash.out;

    // 需要 depth 个哈希和开关子电路用于 Merkle 路径
    component hash[depth];
    component switch[depth];

    // 在每个深度上，将前一个哈希与当前兄弟节点进行哈希，考虑到需要翻转的情况
    for (var i = 0; i < depth; ++i) {

        // 根据兄弟节点的方向设置左右节点 - 如果兄弟节点在左边，需要翻转
        switch[i] = SelectiveSwitch();
        switch[i].in0 <== MerkleTree[i];
        switch[i].in1 <== sibling[i];
        switch[i].s <== direction[i];

        hash[i] = Mimc2();

        // 左节点
        hash[i].in0 <== switch[i].out0;

        // 右节点
        hash[i].in1 <== switch[i].out1;

        // 将左右节点哈希在一起，生成路径上的下一个哈希
        MerkleTree[i+1] <== hash[i].out;
    }

    // 检查顶部哈希 - 应强制为等于 MerkleTree 的根
    MerkleTree[depth] === digest;
}
```

`Spend` 电路使用了 `Merkle` 树的路径上的哈希和开关子电路，通过验证给定的 `Merkle` 证明是否能够证明在 `Merkle` 树中存在指定的 `nullifier` 和 `nonce` 的组合。通过将这些哈希和开关子电

路组合在一起，确保最终的 `intermediate[depth]` 等于输入的 `Merkle` 树根 `digest`，从而验证存在性。

四、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。

```
function computeInput(depth, transcript, nullifier) {
    // TODO
    var tree = new SparseMerkleTree(depth);
    var nonce = null;

    // 将 transcript 编译成树并查找 nullifier
    for (var i = 0; i < transcript.length; i++) {
        // 如果元素个数为1，直接插入树中
        if (transcript[i].length == 1) {
          tree.insert(transcript[i]);
        }
        // 如果元素个数为2
        else if (transcript[i].length == 2) {
          // 检查是不是nullifier
          if (transcript[i][0] == nullifier) {
            nonce = transcript[i][1];
          }
          // 哈希后添加到树中
          tree.insert(mimc2(transcript[i][0],transcript[i][1]));
        }
        else {
          throw("读取 transcript 时出现问题");
        }
    }
    if (nonce == null) {
      throw ("找不到 nullifier");
    }
    var computedInput = {
        digest:     tree.digest,
        nullifier:  nullifier,
        nonce:      nonce,
    };

    var path = tree.path(mimc2(nullifier,nonce));
    for(let i = 0; i < path.length; ++i) {
      const [sibling, direction] = path[i];
      var sibling_string = "sibling[" + i + "]";
      computedInput[sibling_string] = sibling;
      var direction_string = "direction[" + i + "]";
      computedInput[direction_string] = (0 + direction).toString();
    }

    return computedInput;
}
```

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。

五、赎回证明

1. 编译 `spend10.circom`，把结果保存到 `spend10.json`。

```
z@z-virtual-machine:~/Documents/Ex6 (1)/test/circuits$ circom spend10.circom -o spend10.json
```

2. `snarkjs` 将查找并使用 `spend10.json` 进行设置。

```
z@z-virtual-machine:~/Documents/Ex6 (1)/test/circuits$ snarkjs setup -c spend10.json
```

3. 使用 `computeInput.js` 脚本创建输入文件 `input.json`，运行以下命令：

```
z@z-virtual-machine:~/Documents/Ex6 (1)/test/circuits$ node ../../src/compute_spend_inputs.js 10 ../compute_spend_inputs/transcript3.txt 10137
284576094
```

命令行中输入 `computeInput` 的三个参数

- depth = 10
- transcript = transcript3.txt
- nullifier = 10137284576094

生成的 `input.json` 如下：

```
{

"digest":"8080087691978198117050369394765307980410920134807546634402909492739124119015",
    "nullifier":"10137284576094",
    "nonce":"45192935725965",
    "sibling[0]":"171141047017399",
    "direction[0]":"1",

"sibling[1]":"19211042582558215043504006229267785471433863777571347623227723676584345788 01",
    "direction[1]":"1",

"sibling[2]":"16973346385134691586810492335341496079657653339800419377504568909749787593353",
    "direction[2]":"0",

"sibling[3]":"67902546325528562359979961570282748153922526585135067630337700887708167482 13",
    "direction[3]":"1",

"sibling[4]":"13263695242030544851800665282261527384879147831156083700346619397977212257688",
    "direction[4]":"1",

"sibling[5]":"18409670239915602592390322925029059785176147132576777717818708516857552251 71",
    "direction[5]":"0",

"sibling[6]":"40271844808027815520273863391367955085354213880345323634640546178755015059 40",
    "direction[6]":"1",

"sibling[7]":"11615143218180546404420789983788248771891070283082379174961217945780961556045",
    "direction[7]":"0",

"sibling[8]":"69105465193270671129992661212684512462861123951404560429790147319525472108 42",
    "direction[8]":"1",

"sibling[9]":"66914766913649067932887802425774443718038780270602276459541556147550897876 88",
    "direction[9]":"1"
}
```


4. 计算 `witness` 并保存到 `witness.json` 文件中。

```
z@z-virtual-machine:~/Documents/Ex6 (1)/test/circuits$ snarkjs calculatewitness -c spend10.json
```

5. 创建`proof`，命令默认使用`proving_key.json`和`witness.json`文件生成`proof.json`和`public.json`。
   - `proof.json`文件将包含实际的证明。

     按照要求保存到了`artifacts/proof_spend.json`中。
   - `public.json`文件仅包含公共输入和输出的值。

```
test > circuits > {} public.json > ...
1  [
2   "8080087691978198117050369394765307980410920134807546634402909492739124119015",
3   "10137284576094"
4  ]
```

6. 验证`proof`，命令使用`verification_key.json`、`proof.json`和`public.json`进行验证，以确保其有效。在这里，我们正在验证我们知道一个`witness`，其中公共输入和输出与`public.json`文件中的输入相匹配。如果证明有效，输出`OK`，如果无效，则为`INVALID`。

```
z@z-virtual-machine:~/Documents/Ex6 (1)/test/circuits$ snarkjs verify
OK
```

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

六、测试

```
z@z-virtual-machine:~/Documents/Ex6 (1)$ npm test

> cs251-cash@0.1.0 test /home/z/Documents/Ex6 (1)
> 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 (519ms)
    ✓ witness computable for depth 2 (2608ms)
    ✓ witness not computable for bad input (2738ms)


  13 passing (7s)
```

编写的代码全部通过测试。