# 区块链Ex6

2111698于泽林 2112155梁婧涵

## artifacts/writeup.md

```
Answer1:2**i不是信号而是一个常量值。因此sum_of_bits实际上只是输入信号位的线性组合
Answer2:<==运算符是<--和===运算符的组合，其中我们既为信号赋值，又暗示从赋值派生的合约成立。当我们分配信号的线性组合值时，它允许我们避免使用两个运算符。
Answer3:这是无效的，因为由于&运算符，约束不能简化为a*b+c=0(rank-1)的形式
```

## artifacts/verifier_key_factor.json, artifacts/proof_factor.json

在circuit.circom文件中编写了因子分解的算法，如下所示：

```
template Num2Bits(b) {
    signal input in;
    signal output bits[b];
    for (var i = 0; i < b; ++i) {
        bits[i] <-- (in>>i) & 1;
    }
    for (var i = 0; i < b; ++i) {
        bits[i] * (1 - bits[i]) === 0;
    }
    var sum_of_bits = 0;
    for (var i = 0; i < b; ++i) {
        sum_of_bits += (2**i) * bits[i];
    }
    sum_of_bits === in;
}
template SmallOdd(b) {
    signal input in;
    component binaryDecomposition = Num2Bits(b);
    binaryDecomposition.in <== in;
    binaryDecomposition.bits[0] === 1;
}
template SmallOddFactorization(n, b) {
    signal input product;
    signal private input factors[n];
    component smallOdd[n];
    for (var i = 0; i < n; ++i) {
        smallOdd[i] = SmallOdd(b);
        smallOdd[i].in <== factors[i];
    }
    signal partialProducts[n + 1];
    partialProducts[0] <== 1;
    for (var i = 0; i < n; ++i) {
        partialProducts[i + 1] <== partialProducts[i] * factors[i];
    }
    product === partialProducts[n];
}
component main = SmallOddFactorization(3, 8);
```

这段代码用于描述数字分解为奇数因子的过程。

- Num2Bits用于将一个数字转换为二进制
- SmallOdd用于将一个小奇数分解，约束最低位为1
- SmallOddFactorization用于将一个小奇数因子分解，约束乘积等于输入

执行`circom circuit.circom`将该文件编译成电路的约束系统,生成**circuit.json**

执行`snarkjs setup --protocol groth`创建证明和验证密钥，生成**proving_key.json**和**verification_key.json**

创建输入文件**inputs_example.json**，指定要进行因子分解的数，内容如下：

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

执行`snarkjs calculatewitness --circuit circuit.json --input inputs_example.json --witness witness.wtns`运行计算证明的命令，生成证据文件**witness.wtns**

执行`snarkjs proof --witness witness.wtns --proof proof.json --protocal groth`运行生成证明的命令，生成证明文件**proof.json**

执行`cp verification_key.json ../artifacts/verification_key_factor.json`将**verification_key.json**保存到**artifacts/verifier_key_factor.json**中

执行`cp proof.json ../artifacts/proof_factor.json`将**proof.json**保存到**artifacts/proof_factor.json**中

执行`snarkjs verify`验证，结果如下：

```
spider@SILVER-RAT:~/blockchain/Ex6/factor$ snarkjs verify
OK
```

## IfThenElse函数

```
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;
10     signal diff <-- true_value - false_value;
11     out <== condition * diff + false_value;
12 }
```

第9行约束条件**condition**为0或1，确保条件只能取真或者取假

第10行声明辅助信号**diff**，表示条件为真时的值与条件为假时的值的差

第11行根据**condition**的取值使用条件判断表达式计算输出信号**out**的值，如果条件为真则输出**true_value**，否则输出**false_value**

## SelectiveSwitch函数

```
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      s * (1 - s) === 0;
10 
11     component firstOutput = IfThenElse();
12     firstOutput.condition <== s;
13     firstOutput.true_value <== in1;
14     firstOutput.false_value <== in0;
15 
16     component secondOutput = IfThenElse();
17     secondOutput.condition <== s;
18     secondOutput.true_value <== in0;
19     secondOutput.false_value <== in1;
20 
21     out0 <== firstOutput.out;
22     out1 <== secondOutput.out;
23 }
```

该函数实现了选择器，根据输入信号s选择输出in0或者in1

第9行强制s的值为0或者1

下面两部分使用**IfThenElse**来选择输出值，并赋给信号out0和out1

## Spend函数

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

    // TODO
    component computed_hash[depth + 1];

    computed_hash[0] = Mimc2();
    computed_hash[0].in0 <== nullifier;
    computed_hash[0].in1 <== nonce;

    component switches[depth];

    for (var i = 0; i < depth; ++i)
    {
        switches[i] = SelectiveSwitch();
        switches[i].in0 <== computed_hash[i].out;
        switches[i].in1 <== sibling[i];
```

```
        switches[i].s <== direction[i];

        computed_hash[i + 1] = Mimc2();
        computed_hash[i + 1].in0 <== switches[i].out0;
        computed_hash[i + 1].in1 <== switches[i].out1;
    }

    computed_hash[depth].out ===digest;
}
```

该函数实现了计算和验证哈希的功能

定义了一个**computed_hash**数组，用于存储计算出的hash值，对computed_hash[0]进行初始化，使用Mimc2函数计算哈希值。

定义了一个**switches**数组，用于存储SelectiveSwitch组件，通过for循环对switches数组进行初始化，并将computed_hash[i].out和sibling[i]作为输入，direction[i]作为选择信号s传入SelectiveSwitch组件中，然后将该组件的输出作为computed_hash[i+1]的输入。

最后通过判断computed_hash[depth].out是否等于digest来验证计算出的哈希值是否与输入的 digest相匹配。

## compute_spend_input.js

```
function computeInput(depth, transcript, nullifier) {
    // TODO
    const tree = new SparseMerkleTree(depth);
    let input_commitment, input_nonce = [null, null];
    for (let i = 0; i < transcript.length; ++i)
    {
      const commitment_or_info = transcript[i];
      let commitment = null;
      if (commitment_or_info.length == 1)
      {
        commitment = commitment_or_info[0];
      }
      else if (commitment_or_info.length == 2)
      {
        const [t_nullifier, nonce] = commitment_or_info;
        commitment = mimc2(t_nullifier, nonce);
        if (t_nullifier == t_nullifier)
        {
          if (input_commitment != null)
          {
            throw "error";
          }
          [input_commitment, input_nonce] = [commitment, nonce];
        }
        else
        {
          throw "Transcript is invalid"
        }
        if (commitment == null)
        {
          throw "null commitment!"
```

```
        }
        tree.insert(commitment);
      }
    }
    if (input_commitment == null)
    {
      throw "nullifier not found in out transcript";
    }
    const path = tree.path(input_commitment);
    const output = {
      digest:tree.digest,
      nullifier:nullifier,
      nonce:input_nonce,
    };
    for (let i = 0; i < depth; ++i)
    {
      let [s, d] = path[i];
      output['sibling['+i+']'] = s.toString();
      output['direction['+i+']'] = (d) ? "1" : "0";
    }
    return output;
}
```

该函数实现了计算输入的功能，首先创建了一个**SparseMerkleTree**对象tree用于存储Merkle树，根据**transcript**每个元素的不同情况进行不同处理：如果长度为1则将其作为**commitment**直接插入Merkle树中，如果长度为2则将其解构为t_nullifier和nonce,然后使用mimc2计算commitment，并检查与给定nullifier是否相等，若相等则赋值给input_commitment和input_nonce。

然后使用Merkle树的path方法获取input_commitment的路径证明，构建一个包含digest、nullifier、nonce以及路径中的sibling和direction的输出对象然后返回

执行`npm test`

```
spider@SILVER-RAT:~/blockchain/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 (774ms)
    ✓ witness not computable for bad input (833ms)


  13 passing (2s)
```

执行`node compute_spend_inputs.js 10 '../test/compute_spend_inputs/transcript3.txt' 10137284576094`得到**input.json**

执行`cp input.json ../test/circuits`将得到的**input.json**粘贴到**test/circuits**目录

执行`circom spend10.circom -o circuit.json`得到**circuit.json**

执行`snarkjs setup`生成验证密钥文件**proving_key.json**和验证密钥文件**verification_key.json**

执行`snarkjs calculatewitness`生成证据文件**witness.wtns**

执行`snarkjs proof`生成证明文件**proof.json**

执行`cp verification_key.json ../../artifacts/verification_key_spend.json`将验证密钥**verification_key.json**保存到**artifacts/verifier_key_spend.json**中

执行`cp proof.json ../../artifacts/proof_spend.json`将证明**proof.json**保存到**artifacts/proof_spend.json**中

执行`snarkjs verify`进行验证

```
spider@SILVER-RAT:~/blockchain/Ex6/test/circuits$ snarkjs verify
OK
```