

区块链基础及应用 实验六

2113620 任鸿宇 2110937 赵康明 计算机科学与技术

一、小组分工

2110937 赵康明：完成2.了解circom和5.计算花费输入和6.赎回证明和实验报告的书写

2113620 任鸿宇：完成3开关电路和消费电路的代码书写

二、核心代码

2.了解circom

2.1 回答writeup.md中的问题

```
1 Name: []
2
3 ## Question 1
4
5 In the following code-snippet from `Num2Bits`, it looks like `sum_of_bits`
6 might be a sum of products of signals, making the subsequent constraint not
7 rank-1. Explain why `sum_of_bits` is actually a linear combination of
8 signals.
9
10 ...
11     sum_of_bits += (2 ** i) * bits[i];
12 ...
13
14 ## Answer 1
15 在这里，bits是一个信号数组，i是一个索引。表达式(2 ** i) * bits[i]表示一个常量值（2的i次方）与信号值bits[i]的乘积。这个乘法操作是一个线性操作，因为它涉及一个常量系数（2 ** i）与一个信号（bits[i]）的相乘。因此，sum_of_bits通过信号的线性组合进行更新，在整个代码执行过程中仍然是信号的线性组合。
16
17 ## Question 2
18
19 Explain, in your own words, the meaning of the `<==` operator.
20
21 ## Answer 2
22
```

23 <== 运算符应该是硬件编程语言中连续赋值，或者也称为不阻塞赋值。

24

25 **## Question 3**

26

27 Suppose you're reading a `circom` program and you see the following:

28

29 ```

30 signal input a;

31 signal input b;

32 signal input c;

33 (a & 1) * b === c;

34 ```

35

36 Explain why this is invalid.

37

38 **## Answer 3**

39

40 逻辑AND操作符执行逻辑布尔运算。它将两个信号的位逐位比较，并生成一个包含逻辑真值（1）和逻辑假值（0）的结果。这使得它不符合秩为1的约束的标准形式，因为秩为1约束要求等式右侧是线性组合，而不是逻辑组合。所以是无效的。

41

2.2 使用SmallOddFactorization 电路为 $7 \times 17 \times 19 = 2261$ 创建一个证明

1. 阅读circuits文件夹下的example.circom的SmallOddFactorization电路，我们发现其的功能如下：

`SmallOddFactorization` 旨在表示将一个乘积分解成多个因子的过程，其中每个因子都是小的且是奇数。我们来详细分析这个模板的关键组成部分：

a. 输入和因子：

- `signal input product`：这是需要分解的乘积。
- `signal private input factors[n]`：一个有 `n` 个因子的数组，这些因子相乘应该等于 `product`。这些因子是私有输入，意味着它们对证明者是已知的，但对验证者是隐藏的。

b. 因子的约束（小且奇数）：

- 该模板使用另一个子电路 `SmallOdd`，用于确保每个因子都是小的且是奇数。为此，它创建了 `n` 个 `SmallOdd` 子电路的实例，每个因子都通过这些子电路进行验证。
- `smallOdd[i] = SmallOdd(b)`：这里，每个 `SmallOdd` 实例都会接收一个因子作为输入，并确保它符合“小且奇数”的约束。`b` 是定义“小”的参数。

c. 乘积的约束：

- 此部分的目的是确保所有因子的乘积等于输入的 `product`。为了实现这一点，模板引入了 `partialProducts` 信号数组来逐步计算乘积。
- `partialProducts[0] <= 1`：初始化乘积序列。
- 对于每个因子 `factors[i]`，都计算 `partialProducts[i + 1] <= partialProducts[i] * factors[i]`，逐步累积乘积。
- 最后，使用 `product === partialProducts[n]` 确保所有因子的乘积等于原始的 `product`

在理解了这个电路的功能和作用后，我们使用这个电路来创建证明并进行验证。

1.首先我们将电路编译成一个名为circuit.json的文件。

```
1 circom example.circom -o circuit.json
```

2.查找使用circuit.json并运行设置

默认情况下 `snarkjs` 将查找并使用 `circuit.json`，设置的输出将是两个文件：
`proving_key.json` 和 `verification_key.json`。此时我们就得到了验证密钥

```
1 snarkjs setup
```

3.创建input.json文件

我们要使用电路完成 $7 \times 17 \times 19 = 2261$ 的证明，因此我们将乘积为2261，因子为7 7 和19。

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

4.计算见证

```
1 snarkjs calculatewitness
```

5.创建证明

```
1 snarkjs proof
```

6.验证证明

```
1 snarkjs verify
```

7.实验结果

```
Default: circuit.json
print constraints
=====

snarkjs printconstraints <options>
Print all the constraints of a given circuit

-c or --circuit <circuitFile>
    Filename of the compiled circuit file generated by circom.

Default: circuit.json

Options:
--version  Show version number          [boolean]
-h, --help Show help                    [boolean]

Copyright (C) 2018 Okims association
This program comes with ABSOLUTELY NO WARRANTY;
This is free software, and you are welcome to redistribute it
under certain conditions; see the COPYING file in the official
repo directory at https://github.com/iden3/circom
● zhaokangming@ubuntu:~/Desktop/Blockchain/Ex6_包含全部依赖库/Ex6/circuits$ snarkjs setup
● zhaokangming@ubuntu:~/Desktop/Blockchain/Ex6_包含全部依赖库/Ex6/circuits$ snarkjs calculatewitness
● zhaokangming@ubuntu:~/Desktop/Blockchain/Ex6_包含全部依赖库/Ex6/circuits$ snarkjs proof
● zhaokangming@ubuntu:~/Desktop/Blockchain/Ex6_包含全部依赖库/Ex6/circuits$ snarkjs verify
OK
```

验证密钥，证明过程均已粘贴至artifacts目录下。

3.开关电路

3.1 IfThenElse

实现了一个简单的条件逻辑功能。根据输入的条件值，如果条件是真（1），输出结果是真值；如果条件是假（0），输出结果是假值。

```
1 template IfThenElse() {
2     signal input condition;          // 条件输入
3     signal input true_value;        // 真值输入
4     signal input false_value;       // 假值输入
5     signal output out;              // 输出
6
7     condition * (1 - condition) == 0; // 确保条件值为 0 或 1。
8
9     signal diff <-- true_value - false_value; // 计算真值和假值的差值。
10    out <== condition * diff + false_value;
```

```
11 }  
12
```

3.2 SelectiveSwitch

函数设计思路如下：

1. 输入信号：

- `select`：选择信号，它是一个输入信号，可以为0或1。
- `in0`：输入信号0，一个输入信号。
- `in1`：输入信号1，另一个输入信号。

2. 输出信号：

- `out0`：输出信号0，根据选择信号 `select` 控制，可以是输入信号0或输入信号1的值。
- `out1`：输出信号1，与 `out0` 相反，如果 `select` 为0，则为输入信号0的值，如果 `select` 为1，则为输入信号1的值。

3. 约束条件：

- `select * (1 - select) == 0;`：这个约束条件确保 `select` 的值只能是0或1。如果 `select` 不等于0也不等于1，就会违反这个约束条件。

4. 两个 if 语句：

- 通过两个条件判断，根据 `select` 的值来确定输出信号的值。
- `firstOutput` 用于输出信号0 (`out0`) 的计算。如果 `select` 为1，则它选择输入信号1 (`in1`)，否则选择输入信号0 (`in0`)。
- `secondOutput` 用于输出信号1 (`out1`) 的计算。如果 `select` 为1，则它选择输入信号0 (`in0`)，否则选择输入信号1 (`in1`)。

5. 输出信号与条件语句的结果相绑定：

- `out0 <= firstOutput.out;` 和 `out1 <= secondOutput.out;` 这两行代码将输出信号与相应的条件语句结果相绑定，确保选择信号 `select` 控制了输出。

```
1 template SelectiveSwitch() {  
2     signal input select;    // 选择信号  
3     signal input in0;    // 输入信号0  
4     signal input in1;    // 输入信号1  
5     signal output out0;   // 输出信号0  
6     signal output out1;   // 输出信号1  
7  
8     // 强制 select 是 0 或 1。  
9     select * (1 - select) == 0;
```

```

10
11 // 使用两个 if 语句确定输出值。
12
13 // 如果 (select == 1) 则为 in1, 否则为 in0
14 component firstOutput = IfThenElse();
15 firstOutput.condition <== select;
16 firstOutput.true_value <== in1;
17 firstOutput.false_value <== in0;
18
19 // 如果 (select== 1) 则为 in0, 否则为 in1
20 component secondOutput = IfThenElse();
21 secondOutput.condition <== select;
22 secondOutput.true_value <== in0;
23 secondOutput.false_value <== in1;
24
25 // 输出信号必须等于 if 语句的结果。
26 out0 <== firstOutput.out;
27 out1 <== secondOutput.out;
28 }

```

4.消费电路

Spend模板函数逻辑伪代码：

```

1 函数 Spend(depth, digest, nullifier, nonce, sibling[], direction[]) {
2    // 声明变量和数据结构
3    声明 computed_hash 数组, 大小为 depth+1
4    声明 switches 数组, 大小为 depth
5
6    // 计算根据 nullifier 和 nonce 计算第 0 级的哈希值
7    computed_hash[0] = Mimc2(nullifier, nonce)
8
9    // 设置哈希路径上的开关和计算哈希值
10   对于 i 从 0 到 depth-1:
11       switches[i] = 创建 SelectiveSwitch 组件
12       如果 direction[i] 为 true:
13           switches[i].in0 = computed_hash[i].out
14           switches[i].in1 = sibling[i]
15       否则:
16           switches[i].in0 = sibling[i]
17           switches[i].in1 = computed_hash[i].out
18       计算 computed_hash[i + 1] = Mimc2(switches[i].out0, switches[i].out1)
19
20   // 验证最终哈希值是否与给定的 digest 匹配
21   如果 computed_hash[depth].out 等于 digest:

```

```
22     返回验证成功
23 否则:
24     返回验证失败
25 }
```

根据伪代码写出的最终代码:

```
1
2 template Spend(depth) {
3     signal input digest;           // 输入参数, 用于验证最终哈希值
4     signal input nullifier;        // 输入参数, 空化值
5     signal private input nonce;    // 私有输入参数, 随机数
6     signal private input sibling[depth]; // 私有输入参数, 哈希路径的兄弟节点数组
7     signal private input direction[depth]; // 私有输入参数, 指示路径方向的数组
8
9     component computed_hash[depth + 1]; // 存储每个级别中计算的证明哈希值的数组
10
11     computed_hash[0] = Mimc2();        // 第 0 级只是 H(`nullifier`, `nonce`)。
12     computed_hash[0].in0 <== nullifier;
13     computed_hash[0].in1 <== nonce;
14
15     component switches[depth];         // 存储路径中的开关。
16
17     // 设置沿着证明路径的约束。
18     for (var i = 0; i < depth; ++i) {
19         switches[i] = SelectiveSwitch();
20         // 如果 direction[i] 是 true, 我们将计算 H(sibling[i], computed_hash[i])。
21         // 如果是 false, 我们不交换, 计算 H(computed_hash[i], sibling[i])。
22         switches[i].in0 <== computed_hash[i].out;
23         switches[i].in1 <== sibling[i];
24         switches[i].s <== direction[i];
25
26         // 计算下一个级别的哈希值。
27         computed_hash[i + 1] = Mimc2();
28         computed_hash[i + 1].in0 <== switches[i].out0;
29         computed_hash[i + 1].in1 <== switches[i].out1;
30     }
31
32     // 验证 digest 是否匹配最终哈希值。
33     computed_hash[depth].out == digest;
34 }
35
```

函数解释:

1. 输入信号:

- `digest`: 用于验证最终哈希值的输入参数。
- `nullifier`: 用于作废之前的交易或数据的输入参数。
- `nonce`: 一个私有输入参数, 通常是一个随机数, 用于增加哈希的随机性。
- `sibling[depth]`: 私有输入参数, 表示Merkle树中的兄弟节点数组。
- `direction[depth]`: 私有输入参数, 表示构建Merkle证明路径时每一步的方向。

2. 哈希计算:

- 初始哈希 (`computed_hash[0]`) 是通过 `nullifier` 和 `nonce` 计算得出的。使用 `Mimc2()` 函数进行哈希计算
- 对于Merkle树中的每一层 (从 0 到 `depth - 1`), 都会使用 `SelectiveSwitch()` 来根据 `direction[i]` 决定如何结合当前计算的哈希 (`computed_hash[i]`) 和对应层的兄弟节点 (`sibling[i]`) 来计算下一层的哈希值。

3. Merkle证明:

- 通过循环遍历所有层, 该模板构建了一个从叶子到根的Merkle证明。在每一层, 根据 `direction[i]` 的值 (真或假), 选择性地交换 `computed_hash[i]` 和 `sibling[i]` 的位置来计算下一层的哈希。
- 最终, 在达到树的根 (`computed_hash[depth]`) 时, 得到的哈希值应与提供的 `digest` 相匹配。

5. 计算花费电路的输入

函数设计思路:

1. 参数:

- `depth`: 表示稀疏默克尔树 (Sparse Merkle Tree) 的深度, 是树的层数。
- `transcript`: 是一个包含交易历史记录数组, 每个记录可以包含一个或两个元素。
- `nullifier`: 是要查找的目标 nullifier。

2. 稀疏默克尔树的创建:

- 在函数开始时, 创建了一个空的稀疏默克尔树, 它将用于存储和验证承诺值。

3. 遍历交易历史:

- 函数开始遍历交易历史记录数组 `transcript` 中的每个交易信息。
- 对于每个交易信息, 它检查信息的长度, 以确定是直接的承诺值还是需要计算承诺值。
- 如果交易信息长度为 1, 表示这是一个直接的承诺值, 直接使用它。
- 如果交易信息长度为 2, 表示需要计算承诺值, 它使用 `computeMimc2` 函数计算承诺值。同时, 如果目标 nullifier 匹配当前计算的 nullifier, 它记录下对应的承诺值和随机数。

4. 插入承诺值:

- 对于每个承诺值，它将其插入到稀疏默克尔树中。

5. 获取证明路径:

- 如果成功找到了目标 nullifier 对应的承诺值，它将使用稀疏默克尔树的 `path` 方法获取目标承诺值的证明路径。

6. 构建输出对象:

- 最后，函数构建一个包含以下信息的输出对象：
 - `digest`: 稀疏默克尔树的根哈希 (Merkle Root Digest)。
 - `nullifier`: 输入的目标 nullifier。
 - `nonce`: 目标 nullifier 对应的随机数。
 - `sibling[i]`: 第 i 层证明路径中的兄弟节点的字符串表示。
 - `direction[i]`: 第 i 层证明路径中的方向 (0 表示左子树, 1 表示右子树) 的字符串表示。

7. 返回输出对象:

- 最后，函数返回构建的输出对象，包含了证明所需的信息。


```
1 function computeInput(depth, transcript, nullifier) {
2   // 创建一个新的 Sparse Merkle Tree。
3   const merkleTree = new SparseMerkleTree(depth);
4
5   // 存储与目标 nullifier 相关的承诺值和随机数。
6   let targetCommitment, targetNonce = [null, null];
7
8   // 遍历交易历史记录，将它们添加到 Merkle 树中。
9   for (let i = 0; i < transcript.length; i++) {
10     const txInfo = transcript[i];
11     let commitment = null;
12
13     // 如果交易信息长度为 1，直接使用承诺值。
14     if (txInfo.length == 1) {
15       commitment = txInfo[0];
16     }
17     // 如果交易信息长度为 2，计算承诺值。
18     else if (txInfo.length == 2) {
19       const [t_nullifier, nonce] = txInfo;
20       commitment = computeMimc2(t_nullifier, nonce);
21
22       // 检查是否找到匹配的 nullifier。
23       if (nullifier == t_nullifier) {
```

```

24     if (targetCommitment !== null) {
25         throw "不应该有重复的承诺! ";
26     }
27     [targetCommitment, targetNonce] = [commitment, nonce];
28 }
29 } else {
30     throw "交易历史记录无效: " + JSON.stringify(transcript);
31 }
32 if (commitment == null) {
33     throw "出现了空的承诺值! ";
34 }
35 merkleTree.insert(commitment);
36 }
37
38 // 如果我们的目标承诺值没有找到, 抛出错误。
39 if (targetCommitment == null) {
40     throw "在交易历史记录中未找到目标 nullifier";
41 }
42
43 // 获取目标项目的 Merkle 证明路径。
44 const proofPath = merkleTree.path(targetCommitment);
45 const output = {
46     digest: merkleTree.digest,
47     nullifier: nullifier,
48     nonce: targetNonce,
49 };
50
51 // 从证明路径中提取兄弟节点和方向。
52 for (let i = 0; i < depth; i++) {
53     let [s, d] = proofPath[i];
54     output['sibling[' + i + ']'] = s.toString();
55     output['direction[' + i + ']'] = (d) ? "1" : "0";
56 }
57
58 return output;
59 }

```

6. 赎回证明

 使用 circom 和 snarkjs 创建一个 SNARK 用来证明深度为 10 的 Merkle 树中存在与 test/compute_spend_input/transcript3.txt 相对应的 nullifier “10137284576094”。使用深度为 10（你将在 test/circuits/spend10.circom 中找到 Spend 电路的 depth-10 实例化）

6.1 编译电路

在test/circuits下编译电路

```
1 circom spend10.circom -o circuit.json
```

6.2 获取目标 nullifier 对应的 Merkle 证明路径

使用compute_spend_inputs/下的transcript3.txt 相对应的 nullifier “10137284576094 “生成其对应的Merkle路径，作为SNarks的输入数据。

执行指令如下：

```
1 node compute spend inputs.js 10 '../test/compute spend inputs/transcript3.txt'
  10137284576094
```

执行完毕后会生成对应的input.json，我们将其复制到../test/circuits下，作为电路的输入数据

```
1 {"digest":"80800876919781981170503693947653079804109201348075466344029094927391
  24119015","nullifier":"10137284576094","nonce":"45192935725965","sibling[0]":"1
  71141047017399","direction[0]":"1","sibling[1]":"192110425825582150435040062292
  6778547143386377757134762322772367658434578801","direction[1]":"1","sibling[2]"
  : "16973346385134691586810492335341496079657653339800419377504568909749787593353
  ","direction[2]":"0","sibling[3]":"67902546325528562359979961570282748153922526
  58513506763033770088770816748213","direction[3]":"1","sibling[4]":"132636952420
  30544851800665282261527384879147831156083700346619397977212257688","direction[4
  ]":"1","sibling[5]":"1840967023991560259239032292502905978517614713257677771781
  870851685755225171","direction[5]":"0","sibling[6]":"40271844808027815520273863
  39136795508535421388034532363464054617875501505940","direction[6]":"1","sibling
  [7]":"1161514321818054640442078998378824877189107028308237917496121794578096155
  6045","direction[7]":"0","sibling[8]":"6910546519327067112999266121268451246286
  112395140456042979014731952547210842","direction[8]":"1","sibling[9]":"66914766
  91364906793288780242577444371803878027060227645954155614755089787688","directio
  n[9]":"1"}
```

6.3 运行设置

```
1 snarkjs setup
```

6.4.计算见证

```
1 snarkjs calculatewitness
```

6.5创建证明

```
1 snarkjs proof
```

6.6验证证明

```
1 snarkjs verify
```

6.7 实验结果展示

```
Input:json Ex6/test/circuits
() public.json Ex6/test/circuits
包含全部依赖库 [SSH: 192.168.110.134]
snarkjs-0.1.11
src
test
├── circuits
() circuit.json
├── if_then_else.circom
() input.json
├── mimic_cipher.circom
├── mimic.circom
() proof.json
() proving_key.json
() public.json
├── selective_switch.circom
├── sha256_2.circom
├── spend0.circom
├── spend4.circom
├── spend10.circom
├── spend25.circom
() verification_key.json
() witness.json
├── compute_spend_inputs
at Module.load (internal/modules/cjs/loader.js:653:32)
at tryModuleLoad (internal/modules/cjs/loader.js:593:12)
at Function.Module._load (internal/modules/cjs/loader.js:585:3)
at Function.Module.runMain (internal/modules/cjs/loader.js:831:12)
at startup (internal/bootstrap/node.js:283:19)
zhaokangming@ubuntu:~/Desktop/Blockchain/Ex6_包含全部依赖库/Ex6/src$ node compute_spend_inputs.js 10 ../test/compute_spend_input/transcript3.txt
fs.js:114
  throw err;
  ^
Error: ENOENT: no such file or directory, open '../test/compute_spend_input/transcript3.txt'
at Object.openSync (fs.js:443:3)
at Object.readFileSync (fs.js:343:35)
at Object.<anonymous> (/home/zhaokangming/Desktop/Blockchain/Ex6_包含全部依赖库/Ex6/src/compute_spend_inputs.js:131:12)
at Module._compile (internal/modules/cjs/loader.js:778:30)
at Object.Module._extensions..js (internal/modules/cjs/loader.js:789:10)
at Module.load (internal/modules/cjs/loader.js:653:32)
at tryModuleLoad (internal/modules/cjs/loader.js:593:12)
at Function.Module._load (internal/modules/cjs/loader.js:585:3)
at Function.Module.runMain (internal/modules/cjs/loader.js:831:12)
at startup (internal/bootstrap/node.js:283:19)
zhaokangming@ubuntu:~/Desktop/Blockchain/Ex6_包含全部依赖库/Ex6/src$ node compute_spend_inputs.js 10 ../test/compute_spend_inputs/transcript3.tx
t' 10137284576094
zhaokangming@ubuntu:~/Desktop/Blockchain/Ex6_包含全部依赖库/Ex6/src$ cd ..
zhaokangming@ubuntu:~/Desktop/Blockchain/Ex6_包含全部依赖库/Ex6$ cd test/
zhaokangming@ubuntu:~/Desktop/Blockchain/Ex6_包含全部依赖库/Ex6/test$ ls
circuits compute_spend_inputs.js if_then_else.js mimic.js selective_switch.js sparse_merkle_tree.js spend.js util.js
zhaokangming@ubuntu:~/Desktop/Blockchain/Ex6_包含全部依赖库/Ex6/test$ cd circuits/
zhaokangming@ubuntu:~/Desktop/Blockchain/Ex6_包含全部依赖库/Ex6/test/circuits$ snarkjs calculatewitness
zhaokangming@ubuntu:~/Desktop/Blockchain/Ex6_包含全部依赖库/Ex6/test/circuits$ snarkjs proof
zhaokangming@ubuntu:~/Desktop/Blockchain/Ex6_包含全部依赖库/Ex6/test/circuits$ snarkjs verify
OK
```

7.测试

测试结果如下，已通过所有测试。

```
zhaokangming@ubuntu:~/Desktop/Blockchain/Ex6_包含全部依赖库/Ex6/circuits$ npm test
```

```
> cs251-cash@0.1.0 test /home/zhaokangming/Desktop/Blockchain/Ex6_包含全部依赖库/Ex6
> 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 (1333ms)
- ✓ witness not computable for bad input (882ms)

13 passing (3s)