

# 区块链基础及应用实验报告

## Ex6 Tornado-zkSNARK

### Group 31

2112492 刘修铭 2112362 张润哲

## 一、实验要求

探索私有事务 (private transactions) 的实现:

- 制作一个简单版本的花费 Tornado 电路;
- 生成赎回 Tornado 的有效性证明。

## 二、实验过程

### (一) 前期准备

按照实验说明, 进行实验的环境配置。安装完成后, 运行 `npm test`, 可以看到部分样例测试通过, 说明环境配置成功。

```
up to date in 41m
lxmliu2002@lxmliu2002-Ubuntu:~/block/Ex6$ npm test

> cs251-cash@0.1.0 test
> mocha -s is -t 5s test/if_then_else.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 (118ms)
10 failing
```

### (二) 了解 circom

阅读给定的 `example.circom` 电路示例, 了解 circom 的语法等内容, 同时完成 `writeup.md` 中的问题。

- `writeup.md` 已附于附件中。

接着按照 `TUTORIAL.md` 文件的指导, 使用 `SmallOddFactorization` 电路为  $7 \times 7 \times 17 \times 19 = 2261$  创建一个证明。

- 编写 circom 电路, 并编译将其生成为 json 文件;

- 创建可信设置，得到 verification\_key.json 文件与 proving\_key.json 文件。
- 使用命令查看输入输出格式：

```
1xmliu2002@1xmliu2002-Ubuntu:~/block/Ex6/circuits/example$ snarkjs info -c circuit.json
# Wires: 32
# Constraints: 23
# Private Inputs: 3
# Public Inputs: 1
# Outputs: 0
```

- 按照格式，将输入写入到 input.json 文件中：

```
1 {
2     "product": 2261,
3     "factors": [
4         7,
5         17,
6         19
7     ]
8 }
```

- 生成见证文件 witness.json;
- 生成证据，得到 proof.json 与 public.json（公开）；
- 使用 `snarkjs verify` 命令进行验证。

## （三）开关电路

### 1. IfThenElse

IfThenElse 电路验证了条件表达式的正确求值，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；

对于上述要求，使用简单的 if - else 即可实现。具体代码实现如下：

```
1 template IfThenElse() {
2     signal input condition;
3     signal input true_value;
4     signal input false_value;
5     signal output out;
6
7     condition * (1 - condition) == 0; // 保证 condition 为 0 或 1
8     if(condition) out <== true_value;
9     else out <== false_value;
10 }
```

## 2. SelectiveSwitch

SelectiveSwitch 接受两个数据输入 (in0、in1) 并生成两个输出。如果 select(s) 输入为 1, 则它会反转输出中的输入顺序。如果 s 为 0, 则保留输入顺序。该要求同样可借助 if - else 语句实现。具体代码实现如下:

```
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      s * (1 - s) == 0; // 保证 s 为 0 或 1
9      if(s)
10     {
11         out0 <== in1;
12         out1 <== in0;
13     }
14     else
15     {
16         out0 <== in0;
17         out1 <== in1;
18     }
19 }
```

## (四) 消费电路

按照实验手册要求, 该电路应验证所提供的 (私有) Merkle 路径是硬币 H(nullifier, nonce) 的有效 Merkle 证明, 而其根节点的哈希值即为其输入 digest。实现思路已添加到注释中。

```
1  template Spend(depth) {
2      signal input digest; // 根节点 (公开)
3      signal input nullifier; // the nullifier (公开)
4      signal private input nonce; // the nonce (私有)
5      signal private input sibling[depth];
6      signal private input direction[depth];
7
8      // TODO
9      component Hash[depth + 1]; // 保存每一层节点的 hash
10     component switches[depth]; // 根据 direction 调整输出
11     // 初始化 Hash[0] 为硬币 H(nullifier, nonce)
12     Hash[0] = Mimc2();
13     Hash[0].in0 <== nullifier;
14     Hash[0].in1 <== nonce;
15     for (var i = 0; i < depth; ++i)
16     {
17         switches[i] = SelectiveSwitch();
18         switches[i].in0 <== Hash[i].out;
19         switches[i].in1 <== sibling[i];
20         switches[i].s <== direction[i];
```

```

21     Hash[i + 1] = Mimc2();
22     // 左节点
23     Hash[i + 1].in0 <== switches[i].out0;
24     // 右节点
25     Hash[i + 1].in1 <== switches[i].out1;
26 }
27 // 验证是否匹配
28 Hash[depth].out === digest;
29 }

```

## (五) 计算花费电路的输入

1. 借助给定的 SparseMerkleTree 类创建一个新的 SparseMerkleTree，其深度为给定的 depth。

```

1 | let merkleTree = new SparseMerkleTree(depth);

```

2. 遍历 transcript 列表，构建 Merkle Tree。

- 如果数组只有一个元素，表示其要插入 Merkle 树的硬币。
- 如果数组将有两个元素，需要计算其哈希并插入 Merkle 树。

```

1 | for (let i = 0; i < transcript.length; i++)
2 | {
3 |     const item = transcript[i];
4 |     if (item.length === 1) merkleTree.insert(item[0]);
5 |     else
6 |     {
7 |         const nullifierHash = mimc2(item[0], item[1]);
8 |         merkleTree.insert(nullifierHash);
9 |         if (item[0] === nullifier) nonce = item[1];
10 |     }
11 | }

```

3. 创建一个包含 digest、nonce 和 nullifier 的对象，作为 Spend 电路的输入。

```

1 | let spendInput = {
2 |     digest: merkleTree.digest,
3 |     nonce: nonce,
4 |     nullifier: nullifier
5 | };

```

4. 获取 Merkle 路径，将路径上的节点添加到前面创建的对象中并返回。

```

1 let path = merkleTree.path(mimc2(nullifier, nonce));
2
3 for (let i = 0; i < path.length; i++)
4 {
5     const item = path[i];
6     spendInput["sibling[" + i + "]"] = String(item[0]);
7     spendInput["direction[" + i + "]"] = String(item[1] ? 1 : 0);
8 }
9
10 return spendInput;

```

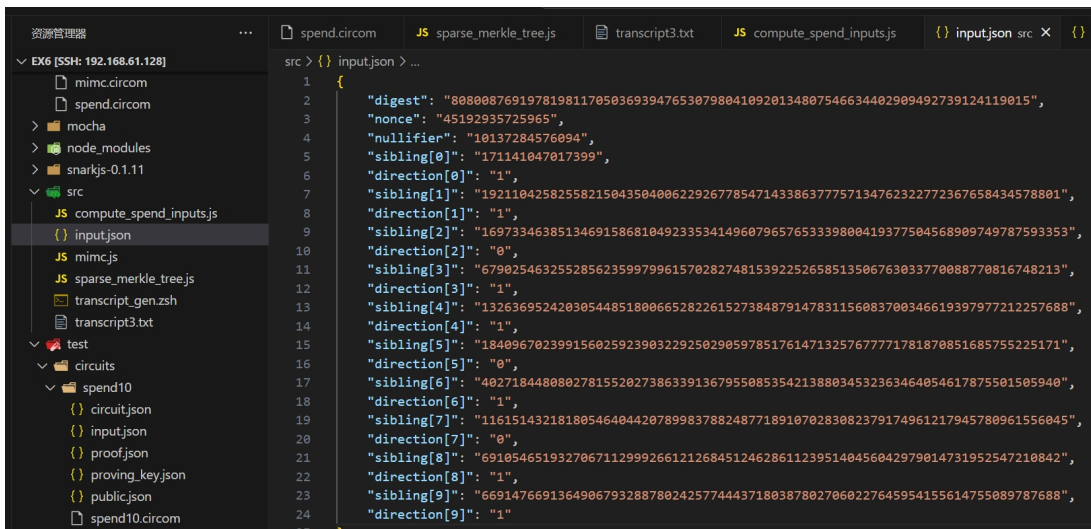
## (六) 赎回证明

按照实验手册说明，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 transcript3.txt 相对应的 nullifier“10137284576094”。

编写好二 (五) 花费电路的输入函数后，借助 compute\_spend\_input.js 文件中的注释，使用以下命令生成 input.json 文件。

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

代码文件中的注释缺少 `node`，对于不熟悉 js 脚本执行的同学存在一定误区。



## 三、实验结果

### (一) 了解 circom

使用命令验证证明，使用之前导出的文件 verify\_key.json、proof.json 和 public.json 来检查证明是否有效。

```
123202872628637544274182200208017171849102093287904247808main.factors[1] +22main.s
bits[3] +88main.smallOdd[1].binaryDecomposition.bits[4] +1616main.smallOdd[1].bina
main.smallOdd[1].binaryDecomposition.bits[7] ] - [ ] = 0
[ 1main.smallOdd[1].binaryDecomposition.bits[2] ] * [ 1 -1main.smallOdd[1].binary
[ 1main.smallOdd[1].binaryDecomposition.bits[3] ] * [ 1 -1main.smallOdd[1].binary
[ 1main.smallOdd[1].binaryDecomposition.bits[4] ] * [ 1 -1main.smallOdd[1].binary
[ 1main.smallOdd[1].binaryDecomposition.bits[5] ] * [ 1 -1main.smallOdd[1].binary
[ 1main.smallOdd[1].binaryDecomposition.bits[6] ] * [ 1 -1main.smallOdd[1].binary
[ 1main.smallOdd[1].binaryDecomposition.bits[7] ] * [ 1 -1main.smallOdd[1].binary
[ 10944121435919637611123202872628637544274182200208017171849102093287904247808 -
in.factors[2] -2main.smallOdd[2].binaryDecomposition.bits[2] -4main.smallOdd[2].bi
n.smallOdd[2].binaryDecomposition.bits[5] -32main.smallOdd[2].binaryDecomposition.
37611123202872628637544274182200208017171849102093287904247807 +109441214359196376
123202872628637544274182200208017171849102093287904247808main.factors[2] +22main.s
bits[3] +88main.smallOdd[2].binaryDecomposition.bits[4] +1616main.smallOdd[2].bina
main.smallOdd[2].binaryDecomposition.bits[7] ] - [ ] = 0
[ 1main.smallOdd[2].binaryDecomposition.bits[2] ] * [ 1 -1main.smallOdd[2].binary
[ 1main.smallOdd[2].binaryDecomposition.bits[3] ] * [ 1 -1main.smallOdd[2].binary
[ 1main.smallOdd[2].binaryDecomposition.bits[4] ] * [ 1 -1main.smallOdd[2].binary
[ 1main.smallOdd[2].binaryDecomposition.bits[5] ] * [ 1 -1main.smallOdd[2].binary
[ 1main.smallOdd[2].binaryDecomposition.bits[6] ] * [ 1 -1main.smallOdd[2].binary
[ 1main.smallOdd[2].binaryDecomposition.bits[7] ] * [ 1 -1main.smallOdd[2].binary
[ -1main.factors[0] ] * [ 1main.factors[1] ] - [ -1main.partialProducts[2] ] = 0
[ -1main.partialProducts[2] ] * [ 1main.factors[2] ] - [ -1main.product ] = 0
● lxmlu2002@lxmlu2002-Ubuntu:~/block/Ex6/circuits/example$ snarkjs setup
● lxmlu2002@lxmlu2002-Ubuntu:~/block/Ex6/circuits/example$ snarkjs calculatewitness
● lxmlu2002@lxmlu2002-Ubuntu:~/block/Ex6/circuits/example$ snarkjs proof
● lxmlu2002@lxmlu2002-Ubuntu:~/block/Ex6/circuits/example$ snarkjs verify
OK
○ lxmlu2002@lxmlu2002-Ubuntu:~/block/Ex6/circuits/example$
```

如图，命令输出 OK，说明证明有效。

## (二) 电路组件测试

按照要求完成开关电路、消费电路及其输入的编写，使用 npm test 命令进行测试。

```
● lxmlu2002@lxmlu2002-Ubuntu:~/block/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 (864ms)
  ✓ witness not computable for bad input (869ms)

13 passing (2s)
```

可以看到测试全部通过，说明函数编写正确。

## (三) 赎回证明

按照 二 (二) 中的流程对其验证：



```
.t2[87] ] = 0
[ -1main.Hash[10].multi.cipher.t2[87] ] * [ 1main.Hash[10].multi.cipher.t2[87] ] - [ -1main.Hash[1
[ -1main.Hash[10].multi.cipher.t4[87] ] * [ 1main.Hash[10].multi.cipher.t2[87] ] - [ -1main.Hash[1
[ -1main.Hash[10].multi.cipher.t6[87] ] * [ 4212716923652881254737947578600828255798948993302968210
10].multi.cipher.t7[86] ] - [ -1main.Hash[10].multi.cipher.t7[87] ] = 0
[ -7594017890037021425366623750593200398174488805473151513558919864633711506220 -1main.switches[9].o
021425366623750593200398174488805473151513558919864633711506220 +11main.switches[9].out0 +11main.Hash
.t2[88] ] = 0
[ -1main.Hash[10].multi.cipher.t2[88] ] * [ 1main.Hash[10].multi.cipher.t2[88] ] - [ -1main.Hash[1
[ -1main.Hash[10].multi.cipher.t4[88] ] * [ 1main.Hash[10].multi.cipher.t2[88] ] - [ -1main.Hash[1
[ -1main.Hash[10].multi.cipher.t6[88] ] * [ 7594017890037021425366623750593200398174488805473151513
10].multi.cipher.t7[87] ] - [ -1main.Hash[10].multi.cipher.t7[88] ] = 0
[ 2908353624093003166282476503660912489227657489563951866097388364093616301216 -1main.switches[9].ou
003166282476503660912489227657489563951866097388364093616301216 +11main.switches[9].out0 +11main.Hash
.t2[89] ] = 0
[ -1main.Hash[10].multi.cipher.t2[89] ] * [ 1main.Hash[10].multi.cipher.t2[89] ] - [ -1main.Hash[1
[ -1main.Hash[10].multi.cipher.t4[89] ] * [ 1main.Hash[10].multi.cipher.t2[89] ] - [ -1main.Hash[1
[ -1main.Hash[10].multi.cipher.t6[89] ] * [ -290835362409300316628247650366091248922765748956395186
10].multi.cipher.t7[88] ] - [ -1main.Hash[10].multi.cipher.t7[89] ] = 0
[ 8286103642026043872859520632100373294886645219515638524788484428425352995084 -1main.switches[9].ou
043872859520632100373294886645219515638524788484428425352995084 +11main.switches[9].out0 +11main.Hash
.t2[90] ] = 0
[ -1main.Hash[10].multi.cipher.t2[90] ] * [ 1main.Hash[10].multi.cipher.t2[90] ] - [ -1main.Hash[1
[ -1main.Hash[10].multi.cipher.t4[90] ] * [ 1main.Hash[10].multi.cipher.t2[90] ] - [ -1main.Hash[1
[ -1main.Hash[10].multi.cipher.t6[90] ] * [ -828610364202604387285952063210037329488664521951563852
10].multi.cipher.t7[89] ] - [ -1main.digest +22main.switches[9].out0 +11main.switches[9].out1 ] = 0
● lxmlu2002@lxmlu2002-Ubuntu:~/block/Ex6/test/circuits/spend10$ snarkjs setup
● lxmlu2002@lxmlu2002-Ubuntu:~/block/Ex6/test/circuits/spend10$ snarkjs calculatewitness
● lxmlu2002@lxmlu2002-Ubuntu:~/block/Ex6/test/circuits/spend10$ snarkjs proof
● lxmlu2002@lxmlu2002-Ubuntu:~/block/Ex6/test/circuits/spend10$ snarkjs verify
OK
● lxmlu2002@lxmlu2002-Ubuntu:~/block/Ex6/test/circuits/spend10$
```

如图，命令输出 OK，说明证明有效。

## 四、说明

本次实验输出文件均已置于 artifacts 文件夹下。

- proof\_factor.json
- proof\_spend.json
- verification\_key\_factor.json
- verification\_key\_spend.json
- writeup.md

本次实验中，**了解 circom** 部分的全部代码置于 circuits/example 文件夹中。

- circuit.json
- example.circom
- example\_zn.circom
- input.json
- proof.json
- proving\_key.json
- public.json
- verification\_key.json
- witness.json

本次实验中，**赎回证明**部分的全部代码已置于 test/circuits/spend10 文件夹中。

- circuit.json
- input.json
- proof.json
- proving\_key.json
- public.json
- spend10.circom
- verification\_key.json
- witness.json