南開大學
Nankai University

南开大学

计算机学院和网络空间安全学院

《区块链基础及应用》实验报告

Ex6:Tornado-zkSNARK

姓名：陆皓喆 学号：2211044 专业：信息安全

姓名：张泽睿 学号：2213873 专业：信息安全

指导教师：苏明

2024 年 12 月 10 日

目录

1 实验目的

在这个实验里，你会学到 circom，一个描述算术电路的工具 snarkjs，一种用于生成和验证电路满意度的 zk-SNARKs 的工具。

你将使用以下知识来探索私有事务 (private transactions) 的实现：

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

2 实验流程

2.1 配置环境

首先，按照实验的说明，我们直接使用配置好的环境进行实验即可，安装完毕 npm 和 nodejs 后，运行 npm test，结果如下所示：

图 2.1: 配置环境成功

发现有部分样例通过了测试，大部分样例没有通过，说明我们的环境搭建成功！

2.2 了解 circom

2.2.1 artifacts/writeup.md

此处共有三个问题，已经在 writeup.md 中完成了回答，为了方便查阅，我将答案放在下面：

Question 1 此处的 bits[i] 表示的是我们输入的二进制的第 i 位，前面的因数 (2 ** i) 表示的是 2 的 i 次方。这一行代码的目的是，将每一位上的二进制数都乘上对应的权重，然后最后将这些值都加起来，最终我们可以得到二进制数的十进制表示。实际上，**我们的 bits[i] 被定义为一个信号，但是我们的 (2 ** i) 并不是一个信号，他只是一个常数**，对于线性组合来说，它是常数和信号的乘积之和，而这里的 (2 ** i) * bits[i] 就是常数 (2 ** i) 和信号 bits[i] 的乘积。因此，sum_of_bits 是输入位的线性组合，而不是信号的乘积。

Question 2 <== 这个符号实际上是 <–和 === 这两个运算符的结合。首先，它分配了一个值给信号，然后 === 说明了我们在分配的过程中派生的合同成立。**这是一种快捷方式，允许我们在使用两个运算符的时候替换成单独的一个运算符**。举个例子，在 example.circom 中，语句 binaryDecomposition.in <== in 就使用了该语句，意思是将输入信号 in 的值赋给子电路 binaryDecomposition 的输入信号，并同时约束 binaryDecomposition.in 等于 in。

Question 3 我们的 a、b 和 c 都是电路信号，第四行代码对信号 a 做了位运算。而在 circom 中，信号的赋值与约束操作是分开进行的，我们不能直接在赋值语句中使用位运算。所以，我们应该对其进行修改，才能使用。具体的修改方法：**首先，通过 a <== (a & 1) * b 语句来完成我们的位运算，并且将结果乘上 b。接着再使用约束语句 c === a 来完成我们的赋值操作**。

2.2.2 artifacts/proof_factor.json

首先，我们利用命令

```
circom ./circuits/example.circom -o ./artifacts/circuit.json
```

来完成对我们的 circom 电路的编译，编译的结果存放在 artifacts 文件夹下。

由于我们的 snarkjs setup 初始化指令要求 json 文件在主目录下，我们将 circuit.json 拖到主目录路径，再运行 setup，就可以得到 verification_key.json 文件与 proving_key.json 文件了。

图 2.2: setup 初始化

然后我们运行 snarkjs info -c circuit.json 来查看对应的输入输出格式，如下所示：

图 2.3: 查看输入输出格式

然后，我们创建新的文件 input.json，这个文件里面存放着我们需要证明的知识，指定了 factors 数组和 product 的值，如下所示：

```
{
    "product": 2261,
```

```
    "factors": [
        7,
        17,
        19
    ]
}
```

接着，我们使用指令 snarkjs calculatewitness 来进行 witness.json 文件的生成。生成的文件内容如下所示：

```
[
 "1","2261","7","17","19","1","1","1","0","0","0","0","0","1","0",
 "0","0","1","0","0","0","1","1","0","0","1","0","0","0","7","119",
 "2261","1"
]
```

然后我们使用指令 snarkjs proof 来生成对应的证据。运行命令后，成功创建了 proof.json 和 public.json。其中，proof.json 文件之中包含着确实的证据。public.json 文件将会包含公共的输入和公共的输出，此处的公共数据为 2261。

然后我们将验证密钥和证明分别保存到文件夹 artifacts 中，运行 snarkjs verify 来进行验证：

图 2.4: snarkjs verify 成功

发现输出 OK，验证完毕！

2.3 开关电路

2.3.1 IfThenElse

该函数的要求是，我们的 condition 必须是 0 或者 1，如果 condition 是 0 的话，就要输出 false_value；如果 condition 是 1 的话，就要输出 true_value。

我们直接使用最简单的 if else 逻辑就可以实现这个要求，代码部分如下所示：

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

```
    condition * (1-condition) === 0;
    if(condition){
        out<==true_value;
    }
    else{
        out<==false_value;
    }
}
```

其中，我们的 condition * (1-condition) === 0 一句，确保了 condition 的值只能为 0 或者为 1。

2.3.2 SelectiveSwitch

SelectiveSwitch 函数接受两个数据输入 (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，则保留输入的顺序。

我们还是利用 if else 逻辑来解决这个问题。首先，我们需要限制 s 的值为 0 或者 1，这一步和上面的函数相类似，我们只需要使用 s * (1-s) === 0 语句就可以做到约束。

具体的代码如下所示：

```
template SelectiveSwitch() {
    signal input in0;
    signal input in1;
    signal input s;
    signal output out0;
    signal output out1;
    s * (1 - s) === 0;
    if(s){
        out1 <== in0;
        out0 <== in1;
    }
    else{
        out1 <== in1;
        out0 <== in0;
    }
}
```

我们可以直接根据 s 的值，来对我们的输出做不同的操作。当 s 为 1 时，我们就倒过来对其进行赋值；如果 s 为 0 的话，就正着不动即可。

不难验证，通过我们的代码的运行，当 s 为 1 时，第一个组件输出 out0 为 in1；第二个组件输出 out1 为 in0；s 为 0 时，第一个组件输出 out0 为 in0；第二个组件输出 out1 为 in1。

2.4 消费电路

我们所编写的代码如下所示：

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

    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;
}
```

这个函数的目的是验证由 nullifier 和 nonce 生成的特定哈希值（我们称之为 coin），是否为 Merkle 树的一部分。我们通过构建从叶子到树根的路径，并验证其正确性，就可以实现这个功能。

首先解释一下源代码中提供给我们的一些变量。digest 指的是 Merkle 树的根哈希值；nullifier 用于生成硬币哈希值的一部分；nonce 则是硬币哈希的另一部分，是我们的私有输入；sibling[depth] 和 direction[depth] 则定义了从硬币到 Merkle 根的路径。

首先，我们通过 computed_hash[0] = Mimc2() 对硬币的初始哈希值进行了计算，然后为初始哈希赋值了左兄弟以及右兄弟。

然后，进入到我们的循环中。函数通过循环 depth 次，逐层构建从硬币哈希到 Merkle 根的路径。在每个层次上，我们使用我们前面编写的 SelectiveSwitch 函数来确定是否需要交换兄弟节点的位置。direction[i] 值决定了是使用左兄弟（sibling[i] ）还是右兄弟作为当前层的哈希计算输入。在每次迭代中，我们都会通过 Mimc2() 函数计算出新的哈希值（computed_hash[i + 1]）。

然后在循环结束后，我们就计算出了我们的根哈希，也就是 computed_hash[depth]。我们只需要将其与 digest 进行比较即可，就可以验证我们这个硬币到底在不在 Merkle 树当中。

2.5 计算花费电路的输入

代码如下所示：

```
function computeInput(treeDepth, trans, nullif) {
  const smt = new SparseMerkleTree(treeDepth);
  let targetCommitment = null;
  let targetNonce = null;

  for (let i = 0; i < trans.length; i++) {
      const element = trans[i];
      let commit;
      if (element.length === 1) {
          commit = element[0];
      } else if (element.length === 2) {
          const [tNullif, nonce] = element;
          commit = mimc2(tNullif, nonce);
          if (tNullif === nullif) {
              if (targetCommitment!== null) {
                  throw "Duplicate found!";
              }
              targetCommitment = commit;
              targetNonce = nonce;
          }
      } else {
          throw `Invalid Transcript: ${trans}`;
      }
      if (commit === null) {
          throw "Commitment is null!";
      }
      smt.insert(commit);
  }

  if (targetCommitment === null) {
      throw "Nullifier not found in transcript";
  }
  const path = smt.path(targetCommitment);
  const result = {
      digest: smt.digest,
      nullifier: nullif,
      nonce: targetNonce
  };
  for (let i = 0; i < treeDepth; i++) {
```

```
        const [sibling, direction] = path[i];
        result[`sibling[${i}]`] = sibling.toString();
        result[`direction[${i}]`] = direction? "1" : "0";
    }
    return result;
}
```

我们来进行一下分析。

首先是函数的参数分析：

- treeDepth: 树的深度，影响 Merkle 树的结构大小。
- trans: 交易列表，其中每个交易可能是一个包含 1 个或 2 个元素的数组。
- nullif: 用于在交易记录中进行匹配的 “nullifier”，如果找到与之匹配的记录，将记录其承诺（commitment）和随机数（nonce）。

我们首先定义了一些变量，分别为：

- smt: 稀疏 Merkle 树实例。
- targetCommitment: 记录匹配的 nullifier 所对应的承诺。
- targetNonce: 对应的随机数（nonce）。

下面是我们的执行流程：

1. **初始化 SparseMerkleTree**：我们通过 treeDepth 参数创建了一个稀疏的 Merkle 树。
2. **循环遍历交易记录**：对于每个交易元素，我们根据元素的长度来进行处理。
    - 如果交易记录的长度为 1，直接将其作为承诺（commitment）；
    - 如果交易记录的长度为 2，就说明可以解构出 nullifier 和 nonce，我们直接通过哈希函数 mimc2 来生成我们的承诺。如果 nullifier 与目标 nullifier 匹配，我们需要记录下承诺和 nonce 的值；
    - 如果交易元素既不包含 1 个也不包含 2 个元素，就说明不符合我们的规范，直接抛出异常即可。
3. **验证 commitment 是否为 null**：如果 commitment 为 null，抛出异常。
4. **插入到 Merkle 树**：将生成的承诺插入稀疏 Merkle 树中。
5. **验证 nullifier 是否找到**：如果没有找到目标 nullifier 的承诺，抛出异常。
6. **获取 Merkle 树路径**：通过目标承诺获取路径。
7. **构造返回结果**：结果对象包含 Merkle 树的 digest、nullifier 和 nonce，以及树路径的兄弟节点和方向信息。遍历树路径，将每个节点的兄弟节点和方向信息添加到结果中。
8. **返回结果**：最后，返回结果对象 result。

2.6 赎回证明

我们首先按照指导书上的步骤，通过命令

```
node compute_spend_inputs.js 10 '../test/compute_spend_inputs/transcript3.txt'
↪  10137284576094 -o input.json
```

首先生成一个 input.json 文件，如下所示：

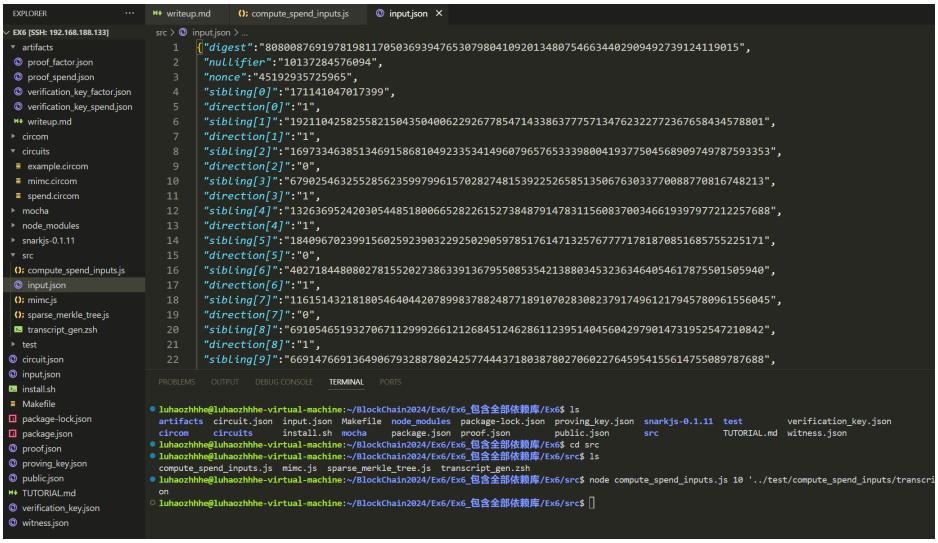

图 2.5: input.json 文件的生成

然后，我们需要将该文件粘贴到 test 的 circuits 目录下，使用 cp input.json ../test/circuits 命令即可。

粘贴完之后，我们直接执行 circom spend10.circom -o circuit.json 命令，在该文件路径下生成了一个新的 circuit.json 文件。

然后，在该路径下，执行初始化操作，即命令 snarkjs setup。生成时间较久，结果如下所示：

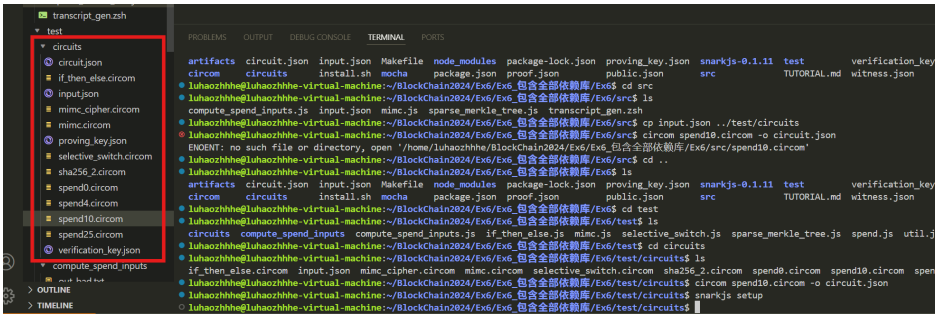

图 2.6: 为 spend10 初始化 snarkjs

这样我们就获得了对应的 proving_key.json 和 verification_key.json。

然后，我们使用指令 snarkjs calculatewitness 来生成对应的 witness.json 即可，如下所示：

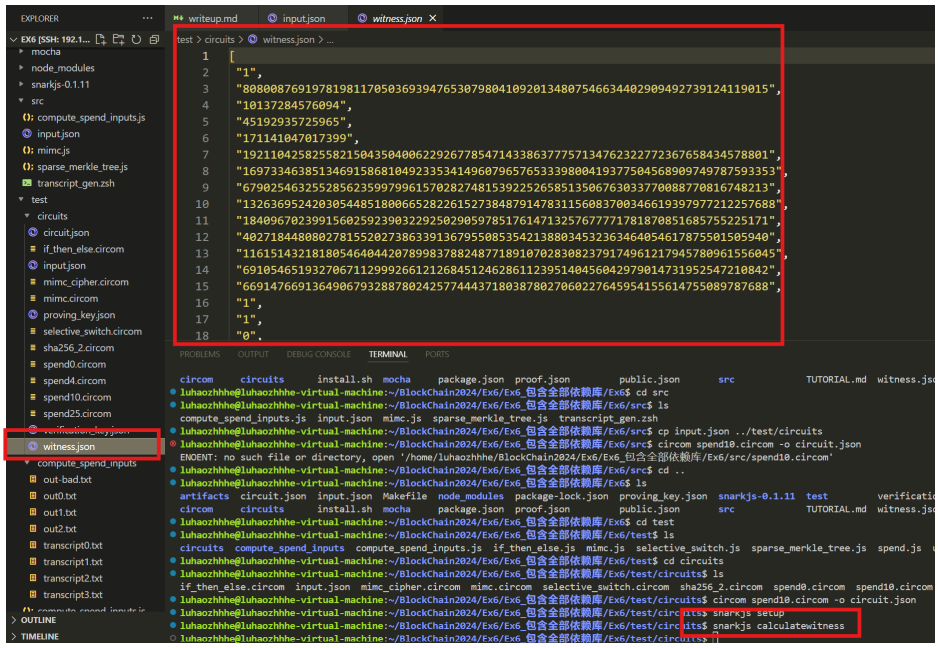

图 2.7: 为 spend10 生成 witness

接着，我们执行 snarkjs proof 命令，来生成对应的 proof.json 文件。

然后，我们将我们生成的 test 文件夹下的 circuits 目录下的 proof.json 粘贴到 artifacts 文件夹中的 proof_spend.json，将 verification_key.json 粘贴到 artifacts 文件夹中的 verification_key_spend.json，通过命令 snarkjs verify 来进行验证，如下所示：

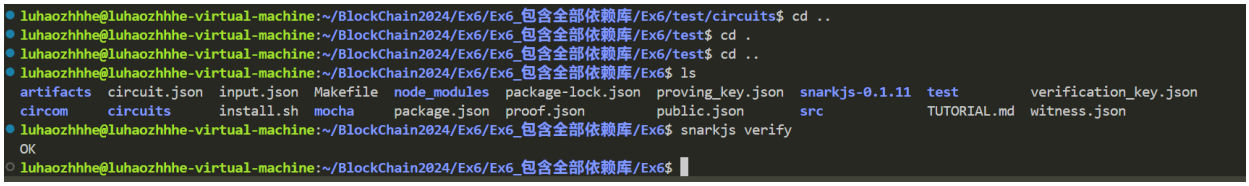

图 2.8: 验证 spend

我们发现，输出 OK，验证完毕！

2.7 测试

最后，我们使用 npm test 命令对我们前面编写的函数做一下检查，结果如下所示：

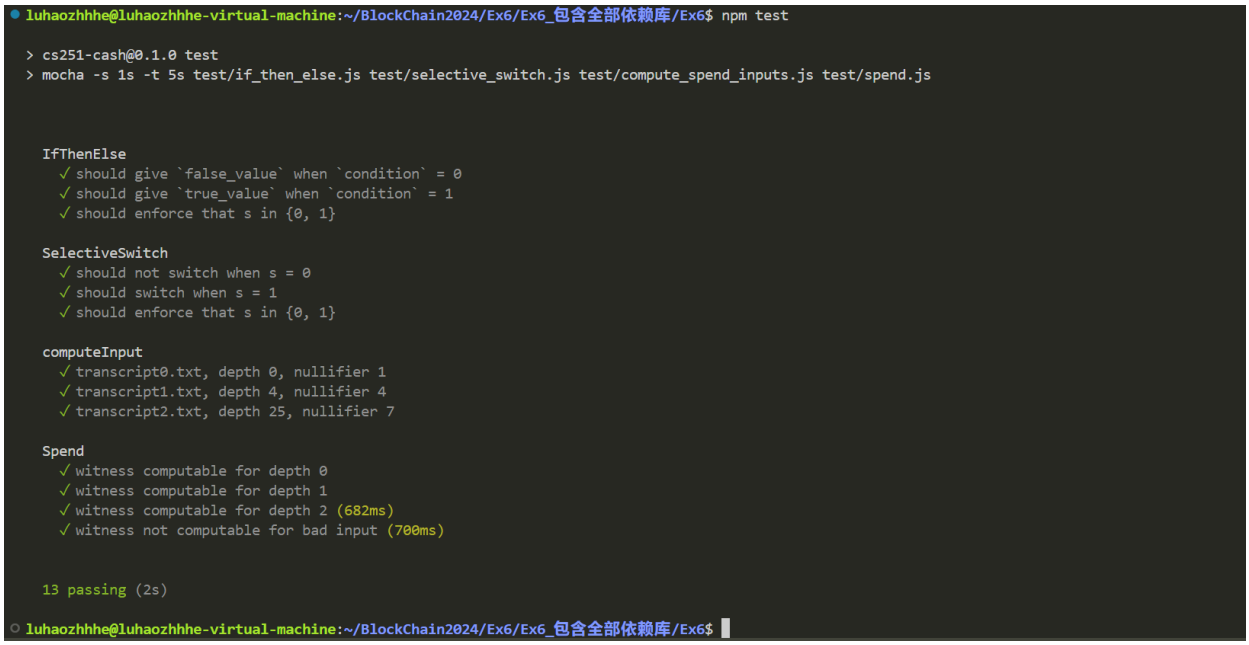

图 2.9: npm test 验证

我们发现，实验未开始前只对了 3 个 point，现在成功通过了所有测试点，说明我们已经完成了简单的 Tornado Spend 电路的设计!