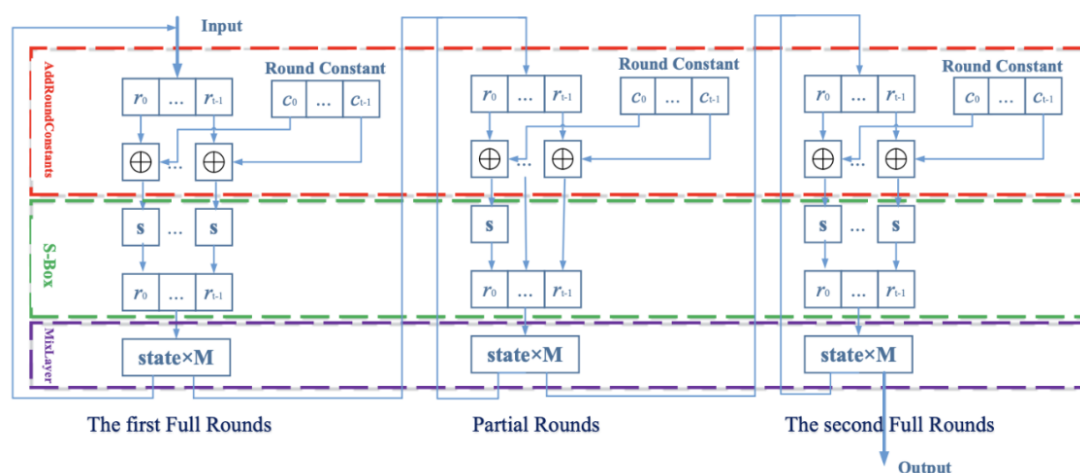


Project03——使用circom实现poseidon2哈希算法的电路

本项目使用circom实现参数为 $(n, t, d) = (256, 3, 5)$ 的poseidon2哈希算法的电路，并用Groth16算法生成证明。

1 poseidon2置换的电路实现

poseidon2的构造如下所示：



其主要分为三步：第一次Full Rounds、Partial Rounds和第二次Full Rounds。Full Rounds中每次循环需要对中间状态向量的所有元素依次完成Add Round Constant，S-Box和MixLayer操作，其中S-Box是一个五次方的模幂运算。Partial Rounds与Full Rounds计算流程基本一致，不同点在于Partial Rounds在S-Box阶段只需完成第一个元素的模幂运算。

- **S-Box:**

```
template SBox() {
    signal input inp;
    signal output out;

    signal x2 <== inp*inp;
    signal x4 <== x2*x2;

    out <== inp*x4;
}
```

- **Partial Rounds**

$t = 3$ 时的内矩阵和外矩阵可以相同，内矩阵的生成方式只需生成三个对角线元素，其余元素为1，该矩阵满足MDS矩阵即可。轮常数每一轮只使用一个，一共需要进行56轮Partial Rounds，因此需要56个轮常数。

```
template InternalRound(i) {
    signal input inp[3];
    signal output out[3];
```

```

var round_consts[56] =
  [ 0x15ce7e5ae220e8623a40b3a3b22d441eff0c9be1ae1d32f1b777af84eea7e38c
    , 0x1bf60ac8bfff0f631983c93e218ca0d4a4059c254b4299b1d9984a07edccfaf0
    ... ...
  ];

component sb = SBox();
sb.inp <== inp[0] + round_consts[i];

out[0] <== 2*sb.out +   inp[1] +   inp[2];
out[1] <==   sb.out + 2*inp[1] +   inp[2];
out[2] <==   sb.out +   inp[1] + 3*inp[2];

}

```

- **Full Round**

Full Rounds中每个状态需要使用一个轮常数，因此每一轮需要三个轮常数。

```

template ExternalRound(i) {
  signal input  inp[3];
  signal output out[3];

  var round_consts[8][3] =

    [ [ 0x2c4c51fd1bb9567c27e99f5712b49e0574178b41b6f0a476cddc41d242cf2b43
      , 0x1c5f8d18acb9c61ec6fcbfcda5356f1b3fdee7dc22c99a5b73a2750e5b054104
      , 0x2d3c1988b4541e4c045595b8d574e98a7c2820314a82e67a4e380f1c4541ba90
      ]
    ... ...
  ];

  component sb[3];
  for(var j=0; j<3; j++) {
    sb[j] = SBox();
    sb[j].inp <== inp[j] + round_consts[i][j];
  }

  out[0] <== 2*sb[0].out +   sb[1].out +   sb[2].out;
  out[1] <==   sb[0].out + 2*sb[1].out +   sb[2].out;
  out[2] <==   sb[0].out +   sb[1].out + 2*sb[2].out;
}

```

完整实现时，最初输入需要进行线性变换，然后进行上述的三个步骤(分别为4轮、56轮、4轮)。

2 poseidon2哈希算法的电路实现

即基于上述置换的海绵结构的哈希算法。

- **参数说明**

t : 状态大小(该项目为3)

capacity: 容量部分大小(1或2)

rate=t-capacity: 速率部分大小(吸收阶段每次处理的元素数量)

- 输入填充

在输入后添加一个1和足够的0使其总长度为rate的倍数:

```
signal padded[padded_len];
for(var i=0; i<input_len; i++) { padded[i] <== inp[i]; }
padded[input_len  ] <== 1;
for(var i=input_len+1; i<padded_len; i++) { padded[i] <== 0; }
```

- 初始化

前 $t-1$ 个元素为0, 最后一个元素为 IV :

```
signal state [nblocks+nout][t  ];
signal sorbed[nblocks  ][rate];

var civ = 2**64 + 256*t + rate;
log("capacity IV = ",civ);

for(var i=0; i<t-1; i++) { state[0][i] <== 0; }
state[0][t-1] <== civ;
```

- 吸收阶段

将填充后的输入分割为多个大小为rate的块, 然后依次处理:

对于每个块, 将块元素与状态的前rate个元素相加, 然后使用poseidon2置换更新状态:

```
for(var m=0; m<nblocks; m++) {

    for(var i=0; i<rate; i++) {
        var a = state [m][i];
        var b = padded[m*rate+i];
        sorbed[m][i] <== a + b;
    }

    absorb[m] = Permutation();
    for(var j=0  ; j<rate; j++) { absorb[m].inp[j] <== sorbed[m][j]; }
    for(var j=rate; j<t  ; j++) { absorb[m].inp[j] <== state [m][j]; }
    absorb[m].out ==> state[m+1];

}
```

- 挤压阶段

从最终状态中提取输出: 首先输出状态的前 $\min(\text{rate}, \text{output_len})$ 个元素, 如果还需要更多输出, 重复应用poseidon2置换并提取前rate个元素:

```
var q = min(rate, output_len);
for(var i=0; i<q; i++) {
    state[nblocks][i] ==> out[i];
}
var out_ptr = rate;
```

```

for(var n=1; n<nout; n++) {
    squeeze[n-1] = Permutation();
    squeeze[n-1].inp <== state[nblocks+n-1];
    squeeze[n-1].out ==> state[nblocks+n ];

    var q = min(rate, output_len-out_ptr);
    for(var i=0; i<q; i++) {
        state[nblocks+n][i] ==> out[out_ptr+i];
    }
    out_ptr += rate;
}

```

3 编译电路

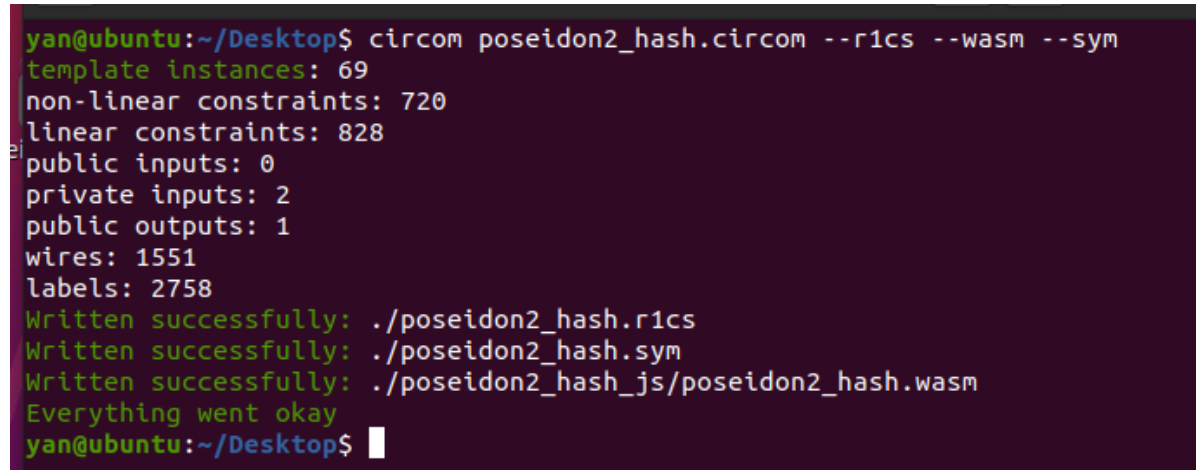
使用下面命令编译电路：

```

circom poseidon2_hash.circom --r1cs --wasm --sym

```

可以看到编译成功：



```

yan@ubuntu:~/Desktop$ circom poseidon2_hash.circom --r1cs --wasm --sym
template instances: 69
non-linear constraints: 720
linear constraints: 828
public inputs: 0
private inputs: 2
public outputs: 1
wires: 1551
labels: 2758
Written successfully: ./poseidon2_hash.r1cs
Written successfully: ./poseidon2_hash.sym
Written successfully: ./poseidon2_hash_js/poseidon2_hash.wasm
Everything went okay
yan@ubuntu:~/Desktop$

```

4 使用Groth16生成证明

- 首先生成可信设置

(1) 下载 Powers of Tau文件

```

wget https://hermez.s3-eu-west-1.amazonaws.com/powersofTau28_hez_final_16.ptau

```

(2) 执行Phase1(Power of Tau)

```

snarkjs powersoftau new bn128 16 pot16_0000.ptau -v
snarkjs powersoftau contribute pot16_0000.ptau pot16_0001.ptau --name="First
contribution" -v

```

(3) 执行Phase2(Circuit-specific)

```
snarkjs powersoftau prepare phase2 pot16_0001.ptau pot16_final.ptau -v
snarkjs groth16 setup poseidon2_hash.r1cs pot16_final.ptau
poseidon2_hash_0000.zkey
snarkjs zkey contribute poseidon2_hash_0000.zkey poseidon2_hash_final.zkey --
name="Contributor" -v
```

(4) 导出验证密钥

```
snarkjs zkey export verificationkey poseidon2_hash_final.zkey
verification_key.json
```

- 然后计算Witness

设置input.json为:

```
{"inp":["123","456"]}
```

生成Witness

```
cd poseidon2_hash_js
node generate_witness.js poseidon2_hash.wasm input.json witness.wtns
```

- 最后使用Groth16生成证明

```
snarkjs groth16 prove poseidon2_hash_final.zkey witness.wtns proof.json
public.json
```

5 验证证明

使用下面命令进行验证:

```
snarkjs groth16 verify verification_key.json public.json proof.json
```

可以发现验证通过:

```
yan@ubuntu:~/Desktop/poseidon2_hash_js$ snarkjs groth16 verify verification_key.
json public.json proof.json
[INFO] snarkJS: OK!
yan@ubuntu:~/Desktop/poseidon2_hash_js$ a
```